International Journal of Software Engineering and Knowledge Engineering

DESIGNING CUSTOMIZED AND TAILORABLE VISUAL INTERACTIVE SYSTEMS

MARIA FRANCESCA COSTABILE[1], DANIELA FOGLI[2], ANDREA MARCANTE[3], PIERO MUSSIO[3], LOREDANA PARASILITI PROVENZA[3], ANTONIO PICCINNO[1]

[1]*Dipartimento di Informatica, Università degli Studi di Bari, via Orabona 4, Bari, 70125, Italy*
costabile@di.uniba.it, piccinno@di.uniba.it
http://ivu.di.uniba.it/people/costabile.htm
[2]*Dipartimento di Elettronica per l'Automazione, via Branze 38, Brescia, 25123, Italy*
fogli@ing.unibs.it
[3]*Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, via Comelico 39/41, Milano, 20135, Italy*
marcante@dico.unimi.it, mussio@dico.unimi.it, parasiliti@dico.unimi.it

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

The paper presents a novel participatory approach to the design of customized and tailorable visual interactive systems; it includes end users, as domain experts, in the design team. A design method is described, which leads to two different visual specifications, one suitable for end users and the other suitable for software engineers. It is also shown how this second specification is directly mapped to the implementation architecture, based on XML technology. The discussion is supported by the description of an example in the mechanical engineering domain.

Keywords: Participatory design, End-User Development, Design specification, XML, specification languages.

1. Introduction

The continuous evolution of technology is determining a parallel evolution of work organizations, creating new research possibilities and challenges for the design and implementation of interactive systems that support people's activities in various work practices. In order to study proper approaches to the design and development of interactive systems, software engineers need an in-depth understanding of the influence of the various problematic aspects characterizing the Human-Computer Interaction (HCI) process. Among these problematic aspects, we primarily consider: the *communication gap between designers and users* [1][2][3], due to the fact that users, HCI experts and designers possess distinct types of knowledge and follow different approaches and reasoning strategies for modeling, performing and documenting the tasks to be carried out in a given application domain; *tool grain* [4], i.e. the system's (or tool's) tendency to push users towards certain behaviors that are often alien to their habits; *implicit*

information [2], embedded in user documents and notations; *tacit knowledge* users have of the application domain [5]; the ample *user diversity*, present even in the same community, as users are different in terms of culture, goals, tasks; the *co-evolution of systems and users* [6] [7] [8]. The latter is of special interest in the overall life cycle of the system, since it is well known that "using the system changes the users, and as they change they will use the system in new ways" [9]. The system must then evolve to satisfy the needs of the evolved user. As a consequence, software engineering paradigms are changing in order to cope with system continuous evolution [10].

Co-evolution also requires a change in the personalization capabilities of interactive systems, so that users can participate in the process by tailoring the system to their needs, i.e. modifying the system in the context of its use, rather than development [11]. This activity results in continuous adaptation of a system and, being performed by the final users of applications, it exploits the potential benefits of task-oriented and skill-based system adaptation that can only be made by end users. In order to tailor their systems, users may create or modify software artifacts, i.e. they may perform End-User Development (EUD) activities, which are currently attracting a lot of attention in many research communities [12] [13].

Because of user diversity, systems designed for specific communities of users must be *customized* by taking into account users' culture, notations, and standard rules [14] and so better supporting people in their specific field of activity. Such systems should also be *tailorable*, to allow end users to adapt them more fully to their needs.

This work presents a novel approach to designing customized and tailorable visual interactive systems. The approach is based on a new conceptual model that exploits the workshop metaphor [15]: in the same way as artisans (joiners, blacksmiths, etc.) keep in their workshops all the tools, and only those, necessary for their specific activities (lathes, mills, etc.), users should find in their software environments, called *Software Shaping Workshops* (SSWs), all the tools, and only those, needed to carry out their activities supported by the computer system. The design of customized systems is achieved thanks to the collaboration of several stakeholders - end users, HCI experts, software engineers -, each contributing to the design, from her/his own point of view. To this end, each stakeholder uses an SSW customized to her/his own culture, skills, background. The system conceptual model yields a network of workshops, some used to perform the activities required to accomplish tasks in the specific domain and work practices, and some used to develop and customize other workshops in the network.

To overcome the communication gap, the conceptual model design is presented in two different visual languages. On one hand, each workshop is specified using a visual language customized to communicate the conceptual model to end users and HCI experts. This visual language is specified in terms of a visual rewriting system, including rules describing how the workshop evolves under the effect of user actions. On the other hand, each workshop is specified using a visual language - the statechart visual language - customized to communicate the conceptual model to software engineering implementers. The sentences in this visual language specify control finite state automata, which are used

to express the rewriting rules in a notation understood by software engineers, who can subsequently translate automata into programs. The two specifications are coherent and complementary and are directly reflected in the implementation architecture that we propose, based on XML technology.

The paper has the following organization. Section 2 provides some considerations about the need for participatory approaches in the design of visual interactive systems; End-User Development features are also emphasized, supporting the co-evolution of users and systems. Section 3 describes the proposed conceptual model based on the workshop metaphor. Section 4 presents the formal basis for system design. Section 5 describes the visual languages adopted to specify the systems for end users, HCI experts and software engineers. Section 6 presents the approach to system implementation. Section 7 discusses the related literature and Section 8 concludes the paper.

2. Recognizing the need for participatory design and End-User Development

The design of an interactive system requires more knowledge than is possessed by a single software engineer or HCI expert. End users, for example, are the "owners of problems", and have a domain-oriented view and a knowledge of the processes to be automated, which must be taken into account in the design. In turn, software engineers have a knowledge of tools and techniques for system development and HCI experts have a knowledge of system usability and human behaviour. They are necessary to the development of the system because they are the only ones who can guarantee the usability and performance of the system. All these experts must contribute their experience to the design and implementation, but no one is more important than the others. This means that the design must be developed by a team that has to include at least software engineers, HCI experts and representatives of end users - called here *domain expert users*. All of them must recognize: 1) that each member of the team complements the ignorance of the others, 2) the need to reach a mutual understanding, and 3) the need for peer collaboration [16].

In this view, end users play two roles: a) as domain experts they reason about their own work activity and design their own work environment; b) as competent practitioners they perform their work activities, finding solutions to their problems in their own domain and possibly adapting virtual tools to their needs. For example, a physician, as a competent practitioner, uses the system to carry out all the activities required to reach a diagnosis in a specific case and, as a domain expert, participates in the design of the interactive system that supports the diagnostic activity [14][15].

Once the system is in use, the design team will observe end user activities, the new usages of the system, the new procedures induced by the evolving organization, and will monitor end user complaints and suggestions about the system. On the basis of these observations, the design team can update the system and sometimes also the underlying software technologies. Co-evolution results, therefore, in a cyclic process, in which usage

of the system induces an evolution in the user culture and organization, which in turn induces an evolution of the system and of the technology.

Facing co-evolution is not an easy task, also because the end user population is not uniform, but includes people with different cultural, educational, training, and employment backgrounds, novice and experienced computer users, the very young and the elderly, people with different types of disabilities. Moreover, these users operate in various interaction contexts and scenarios of use and their aim is to exploit computer systems to improve their work.

End users do not always perform repetitive activities; often, they are required to face unforeseen situations, in which they need to create new procedures and tools or to adapt existing procedures and tools to solve problems that cannot be predicted in advance. End users are increasingly required to be able to produce their own software, developing software artefacts in support of organizational tasks [17]. On the whole, end users need to act as designers in some steps of their activities and as traditional users in others [18].

In literature, end-user programming and end-user computing are often used as interchangeable terms, for example in [19], the authors discuss “enhancing editors with *end-user programming capabilities*”. They also say that “end-user computing is needed in domains or applications where the activity cannot be planned in advance” and that it should have the flavor of “on-the-fly” computing, i.e., it should emerge during the user activity, when the user needs to create a combination/repetition/abstraction construct, in response to some concrete situation. Brancheau and Brown describe *end-user computing* as "the adoption and use of information technology by people outside the information system department, to develop software applications in support of organizational tasks" [17].

An interactive system should be designed to support end users both when they are simply using the interactive system and when they are acting as designers of their software tools. These activities last throughout the life of the interactive system and, therefore, the design team remains active for the whole life of the interactive system. In this view, “End-User Development” denotes the set of methods, techniques and tools that allows end users to create or modify the interactive system whenever necessary and can *support the continuous co-evolution of the system and its users* [8][20].

3. The conceptual model

Our approach to the design of visual interactive systems that are customized to users’ needs, preferences, culture, skills, but also permit users to further tailor them, has originated from observing professional people - mechanical engineers, geologists, physicians - in their work practices. Software Shaping Workshops (SSWs) dedicated to these end users are organized in analogy to an artisan workshop, i.e., a small establishment where an artisan, such as a blacksmith or a joiner, manipulates raw materials in order to manufacture artifacts; to this aim, the artisan equips her/his work bench with only the tools needed.

Following the analogy, an SSW is designed as a *virtual workshop*, in which end users find all the (virtual) tools, and only those, whose shape, behavior and management are familiar to them. In the virtual workshop, a virtual bench is present on which the end user can perform the desired activities. For each activity, s/he can select the tools best suited to the performance. On the whole, the Software Shaping Workshop allows end users to carry out their activities and adapt environment and tools without the burden of using a traditional programming language, but using high level visual languages tailored to their needs. While traditional artisans shape real supplies, end users shape software artifacts. This is the reason why they are called *Software Shaping Workshops*.

To better illustrate the proposed approach, we refer herein to a case study in the mechanical engineering field. The work is the result of our collaboration with a company that produces systems for factory automation, to be sold to the company clients, who use them in their work processes. In this scenario, the company must produce systems which clients may customize according to the specific needs of their workers. The company has the following needs: 1) to create systems for factory automation that are customized to its clients, i.e. easy to learn, to use, and to adapt for experts in a given domain, different from computer science and automation science; 2) to rely on software tools which support the company personnel in the development, testing, and maintenance of such systems. The company personnel is organized in different categories of people with different responsibilities and skills, who need to perform various tasks with the software tools. In our prototype, two software environments for this case study have been developed: one devoted to assembly-line operators who control the automation systems in the factory, and one that permits mechanical engineers to create the software environment for assembly-line operators.

Fig. 1 shows the SSW devoted to an assembly line operator controlling a pick-and-place robot, developed for a case study in the mechanical engineering field. Fig. 1a shows the initial state of the system. The functionalities required by the assembly line operators for such an environment include: different modalities of using the robot (automatic, manual, diagnostic, setting, etc.), the possibility to choose among various tools to be associated with the robot to modify its behavior and the task to perform and, finally, a number of options to create annotations, get online help, save the work, etc. The robot operation modality is chosen by clicking on one of the buttons in the upper button panel in Fig. 1a; the tools to be associated with the robot may be selected from the archives of pieces, engines, trajectories and grippers shown on the right side of the SSW; the behavior of the machine is then shown in the work area in the center. For example, after the button “Automatico” (“Automatic”) has been clicked, the system presents in this area a schematic version of the robot, including the parts to be checked before its activity may start (Fig. 1b). When the user clicks the button “avvia” (“start”), the robot starts in an automatic modality. The user can thus observe in the working area that the engine (“Motore”), the gripper (“Pinza”) and the piece (“Pezzo”) are checked. Finally, at the bottom, a message area presents messages orienting the user during her/his interaction

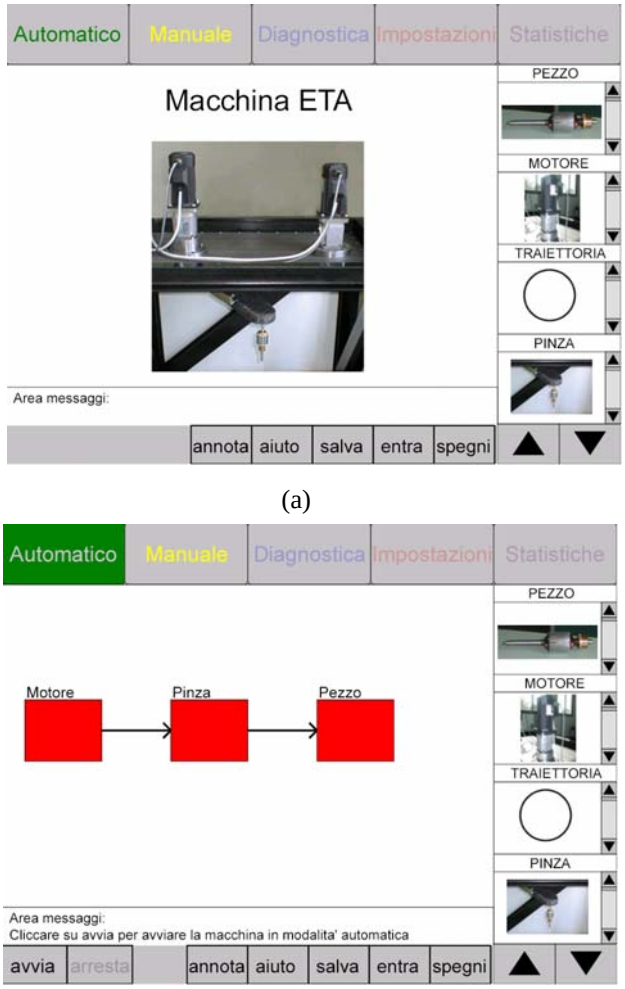

Fig. 1. The SSW devoted to assembly-line operators.

with the system and a button panel offers the options of annotation, help, saving, logging, exiting.

The design methodology derived from the approach based on SSWs requires each SSW to be able to exploit the language and notation adopted by the specific user community to which it is addressed. In order to convey implicit information, data are organized according to the user's culture and work context, and tools are presented by images, texts, or icons, expressive to the users of that particular domain. For example, images of grippers and engines on the right side of figures 1a and 1b are meaningful to practitioners in the automation systems field. Furthermore, each SSW enables actions that

resemble those in the traditional real contexts to be performed. For instance, in the workshop devoted to assembly-line operators, users may push buttons to start the machine and to control its subsequent operation. In this way, users are facilitated by being able to exploit their tacit knowledge while performing their work task.

The methodology emphasizes a meta-design perspective [21], which goes beyond, but includes user-centered design and participatory design [22]. Meta-design, consists in the design of workshops that end user representatives may use to design for themselves the workshops for end users. This can bridge the communication gap between end users, HCI experts and software designers, and support co-evolution. Moreover, involving end users in system design permits the development of software environments that do not speak a computer-oriented language and do not induce tool grains [4]. The users themselves are aware of the needs, background, skills and habits of the community they belong to and may build environments that are more acceptable than those directly created by software engineers. As already mentioned in the previous section, end users play two main roles in the lifecycle of interactive software systems: 1) they perform their work tasks; 2) they participate in the development of software environments as stakeholders of the domain. In the first role, end users interact with a type of SSWs, called *application workshops*. During the use of an application workshop, the users can tailor it according to their preferences and working needs. For example, the SSW devoted to assembly-line operators (Fig. 1) can be tailored to make the robot perform different trajectories or manufacture different products, but also to manage different kinds of robots. In the second role, as members of the design team, end users participate directly in the development and customization of application workshops using different workshops, called *system workshops*. Referring to our case study, Fig. 1 shows the application workshop developed for the assembly line operator through the system

Fig. 2. A screenshot of the system workshop devoted to mechanical engineers.

workshop shown in Fig. 2. Such a SSW is a software environment customized to the culture and background of mechanical engineers (domain experts), which makes it possible both to generate and update or customize application workshops just by direct manipulation, through simple drag-and-drop activities. The screenshot in Fig. 2 was taken during the interaction of a domain expert who was creating the application workshop shown in Fig. 1. The application workshop in the figure is partially composed. The mechanical engineer was positioning a new button ("Diagnostica") on the upper button panel and changing its position by simply moving the visual object.

In the SSW methodology, the concept of the system workshop is general: actually, system workshops are developed to allow the members of each community involved in the design and validation of the system to participate in this activity. For example, system workshops for HCI experts and software engineers are used. Each member of the design team can examine, evaluate and modify an application workshop using tools shaped to her/his culture. In this way, this approach yields a workshop network that aims to overcome the differences in language among the experts of the different disciplines (software engineering, HCI, application domain), who can cooperate to develop computer systems customized to the needs of the user communities.

The SSW network is structured so that the different stakeholders can participate in the application workshop's design, implementation, and use without being disoriented. In general, a network is organized in levels (Fig. 3). At each level, one or more workshops

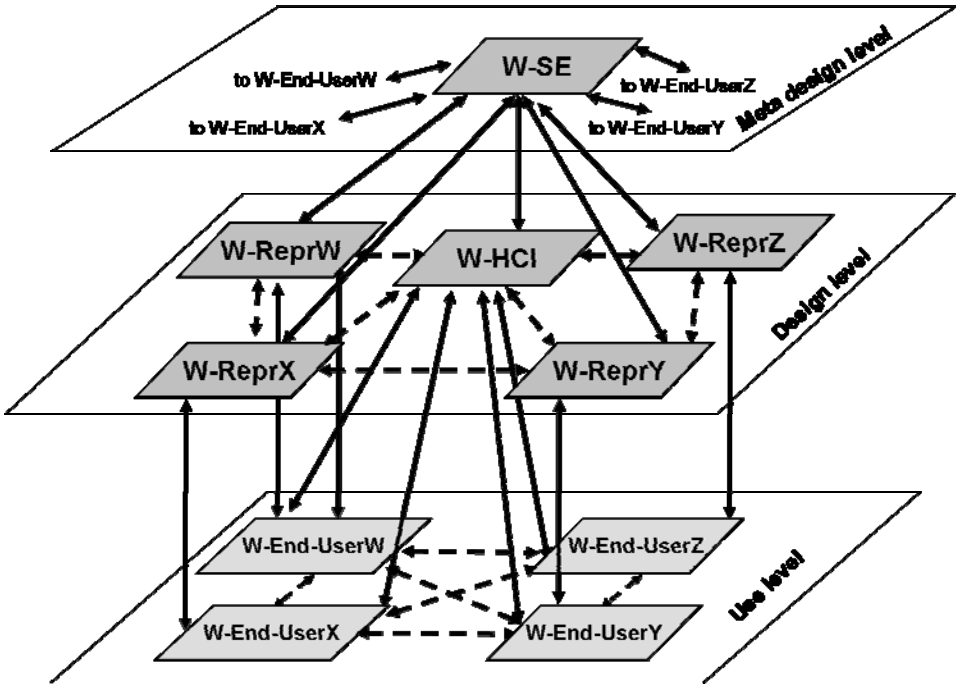

Fig. 3 The network of SSWs.

can be used, which are connected by communication paths. Fig. 3 presents a generic workshop network including three levels:

a) *the meta-design level*, where software engineers use a system workshop, called W-SE, to prepare the tools to be used and to participate in the design, implementation, and validation activities;

b) the *design level*, where HCI experts and domain experts cooperate in the design, implementation, and validation of application workshops; a design member belonging to the community X uses a system workshop W-ReprX, created by the software engineers and customized to the needs, culture and skills of community X; the various experts design customized application workshops and also tailor their own system workshop;

c) the *use level*, where end users cooperate to achieve a task, using application workshops customized to their needs, culture, and skills: end users belonging to the community X participate in task achievement using the application workshop W-End-UserX customized to their needs, culture, and skills [8] [15].

As far as SSW adaptation is concerned, at the meta-design and design levels the customization of SSWs to be used at the next levels is performed. At the use level, end users may tailor their application workshop to their own needs and preferences.

The proposed conceptual model includes all the SSWs organized in a network. On the whole, both meta-design and design levels include all the system workshops that support the design team in performing the activity of participatory design. Such system workshops can be considered User Interface Development Environments (UIDEs) [23]. The novel idea is that the UIDEs used by domain experts are very much oriented to the application domain and have specific functionalities, so that they are easy for domain experts to use.

4. Software Shaping Workshop design

Model-based approaches to HCI attempt to identify a unifying framework - a model - to describe the interaction process. The model is used to identify the causes of usability difficulties affecting interactive systems. An HCI model can thereafter be used to derive design procedures, which allow the implementation of systems in which these difficulties are eliminated or at least reduced. The work presented here is based on the model of the HCI process and on the theory of visual sentences developed in [24]. In that approach, HCI is modelled as a process in which systems of different natures (the cognitive human - the 'mechanical' machine) cooperate to achieve a task. From this point of view, HCI is a process in which user and computer communicate by materializing and interpreting a sequence of messages at successive instants in time. The characterizing aspect of such a HCI model is the recognition of two interpretations of the exchanged messages: one performed by the human based on the human's cognitive criteria, and the other by the computer based on the programs implemented by the designers.

In WIMP interaction, a message exchanged between the user and the computer is the whole image i represented on the computer screen, formed by texts, pictures, icons, etc.

Humans look at the screen and interpret the visual message - the image - currently shown by the computer within the context of their activity, by recognizing *characteristic structure*s, css, i.e., sets of pixels representing functional or perceptual units for the humans (e.g., the button "Automatico" shown in Fig. 1). On the machine side, each characteristic structure, cs, is the physical manifestation of an entity, referred to as a *virtual entity*, which exists because the computer interprets a program P specifying its appearance and behavior. A virtual entity (ve) is a virtual dynamic open system. It is *virtual* in that it exists only as the result of the execution of the program P by a computer; *dynamic* in that its behaviour evolves over time; *open* in that its evolution depends on its interaction with the environment. The program P, whose execution creates and maintains active a virtual entity ve, is composed of a set of programs: some of which, In (Input) programs, acquire the input events for the ve generated by the user actions; some, AP (APplication) programs, compute the ve reactions to these events; and some, Out (output) programs, materialize the ve reactions by showing the characteristic structure representing the new state of the ve to be shown to the user. The program P synthesizes how the designers understand the activities to be performed.

The ve state is defined as a *characteristic pattern* cp=<cs, u, <intcs, matcs>>, where intcs (interpretation) is a function, mapping the current cs of the ve into the current computational state u of the program AP and matcs (materialization) is a function mapping u into cs.

The user interprets a characteristic structure cs (of a ve) within the image i and produces her/his messages by performing an operation on the cs through the input devices available in the computer at hand. On the other side, the computer, captures input events generated by user actions and reacts by providing a visual feedback allowing the human to evaluate the correctness of the action being performed.

An interactive system (an SSW in our terminology) is constituted by virtual entities interacting with one another and with the user through the I/O devices. The user sees the SSW as a whole ve, whose computational state u is materialized at each instant as an image i on the screen. This association can be specified as a triple vs=<i, u, <int, mat>>, where i is the array of pixels constituting the current image, u is a suitable description of the current state of the process determining the reaction of the whole system to user activities, int and mat are two functions relating elements of i with components of u. This triple is called a *visual sentence* (vs) in [24], and specifies the state of the whole virtual entity (i.e. the whole SSW). More details on this model can be found in [25].

5. Design specification

The design activity results in two kinds of specification. The first one exploits a notation (a visual language) understandable by end users (then also by HCI experts), based on the visual elements which should appear on the screen as a consequence of user actions and computer reactions. The second kind of specification exploits another visual language

that conforms to software engineers' experience and background, and it is used as starting point for the implementation activity.

5.1. *Specification for end users and HCI experts*

Being the state of the virtual entity representing a whole SSW a visual sentence, the set of admissible states of a SSW is a set of vss. Visual sentences result from the composition of simpler characteristic patterns, i.e. of the states of the virtual entities composing the SSW.

The design activity aims at specifying the set of vss of a SSW and the transformations between vss. It is thus necessary to define first a finite set (visual alphabet) of cp prototypes, whose instances may be composed to form vss [26]. More precisely, a cp prototype is the initial state of a virtual entity composing the SSW, from which it is possible to obtain all the other states (cps) of the virtual entity as a result of the interaction with the user or with other ves. The visual alphabet is designed in a participatory way by all stakeholders, and is based on user language and notation. Next, a Visual Conditional Attributed Rewriting system (vCARW) [24] must be defined on the visual alphabet. A vCARW system contains a set R of visual rewriting rules, which are used to transform a visual sentence vs_1 into another visual sentence vs_2, by introducing new cps or modifying existing ones.

VCARWs are characterized by rules in which only the pictorial part is made explicit to the design team, that includes end users and HCI experts, besides software engineers. Therefore, the physical appearance – i.e. topology, geometry and shape – of each cs involved in the rewriting step is defined. On the other hand, the computational meaning of the rules is described in a textual form, or by animations or prototypes, while the technical details about the internal representation of the rules are not explicitly discussed at this stage of development.

The rewriting rules are exploited to specify how the interaction process evolves. In each state of the interaction a finite number of user activities can be performed. As a consequence of a user activity, a visual sentence vs_1 is transformed into a visual sentence vs_2. The interaction process is specified as a sequence of such transformations. For example, the visual sentence whose image part is in Fig. 1a is transformed into the visual sentence whose image part is in Fig. 1b, as a consequence of the user clicking the button "Automatico".

In a transformation, vs_1 and vs_2 share a common part, while the variable part of vs_1 is transformed into the variable part of vs_2 through the application of a transformation rule in the form tr: <a, r>, where a is the user activity and r is a rewriting rule in R.

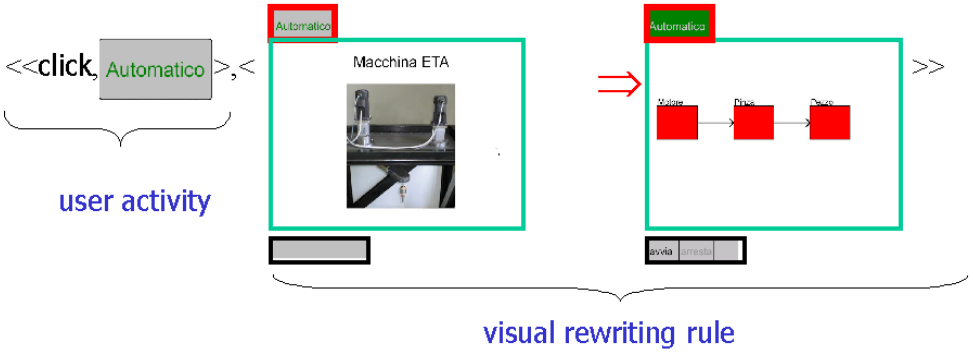

Fig. 4. The pictorial part of a transformation rule.

Fig. 4 shows the pictorial part of the transformation rule that fires when the user clicks the button "Automatico" while interacting with the application workshop shown in Fig. 1. The user activity is specified as a pair a=<op, cs>, where op is a physical operation performed by the user (a mouse click in this case) and cs is the set of pixels to which op is referred. The visual rewriting rule defines which css are transformed from one state to the other: in this case, they are three disjointed sets of pixels denoting the resources required to apply the rule. The computational meaning of the rule may be described as follows: the state of the button "Automatico" changes from non selected to selected; the state of the working area changes in order to present a schematic depiction of the composition of the robot; the state of the button panel under the working area changes by presenting the operators needed to start and check the robot. This actually means that the states of three virtual entities change, and, as a whole, the system reaches a state in which it is possible to start the machine in the automatic modality and observe the ongoing automatic check process.

In summary, each SSW is formally designed in a collaborative way by software engineers, HCI experts and end users by specifying: 1) a *visual alphabet* of cp prototypes; 2) an *initial state*, vs_0, of the workshop, which is instantiated when the user first accesses the system; 3) a set of rewriting rules R; 4) a set of *transformation rules* TR. Even though the computational meaning of transformation rules is also discussed by stakeholders verbally or through prototypes, this kind of specification focuses more on the visual aspects of the system transformations.

5.2. *Specification for software engineers*

The four design components described above can be specified in a program-oriented way by using Control Finite State Machines. We use Harel's statecharts [27] to specify a SSW as a complex virtual entity. The hierarchical structure of an SSW, obtained in terms of ve composition, can be easily specified by statecharts permitting the modeling of systems at different levels of abstraction.

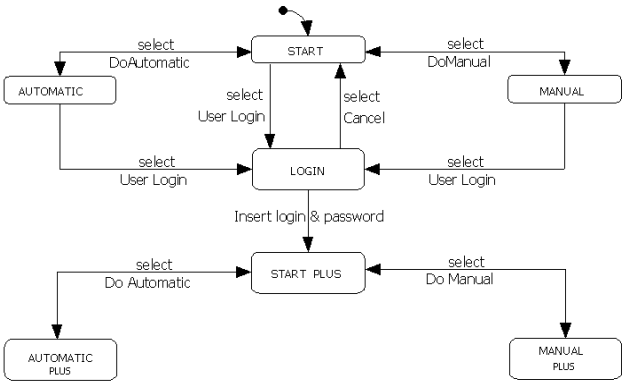

Fig. 5. A portion of the statechart specifying the SSW devoted to assembly-line operators.

In the case of SSWs, according to the level of abstraction adopted for the description, the states of the statechart may represent classes of equivalent vss or instances of vss, which are in turn compositions of cps [25]. For example, Fig. 5 shows a portion of the statechart specifying the application workshop devoted to assembly-line operators at the highest level of abstraction. The states of this statechart are classes of visual sentences: for instance, the state "Login" is the class of the visual sentences associated with the login screen shot. Once both login and password have been filled and "Insert" is selected, a new screen shot appears, changing the class of visual sentences representing the next state of the SSW. The transitions from one state to another occur as a consequence of user activities, namely of operations performed with reference to some css included in the image part of the current visual sentence. For example, in Fig. 5, "select DoAutomatic" refers to a selection activity performed on the button "Automatico", i.e. a mouse click on the cs of this virtual entity.

The transformation rules are thus translated into the transition and output functions of the statechart. In a transformation rule <a, r>, user activity a is the input event firing a transition, r is the rewriting rule to be applied, and thus it specifies which will be the next state in the statechart to be reached (transition function) and which computational activity will be activated (output function) [25].

This kind of specification lays more emphasis on the computational meaning of transformations and easily drives software engineers during the SSW implementation.

6. System implementation

The workshops in a SSW network are implemented as IM^{2}L programs. IM^{2}L (Interaction Multimodal Markup Language) is an XML-based language that provides the rules for the definition of the virtual entities that compose the SSW: its markup tags encode a description of the storage organization and logical structure of the virtual entities. In the following, we call IM^{2}L *fragment* a piece of an XML document identifying a virtual entity, be it elementary or complex. For example, the virtual entity "button" will be

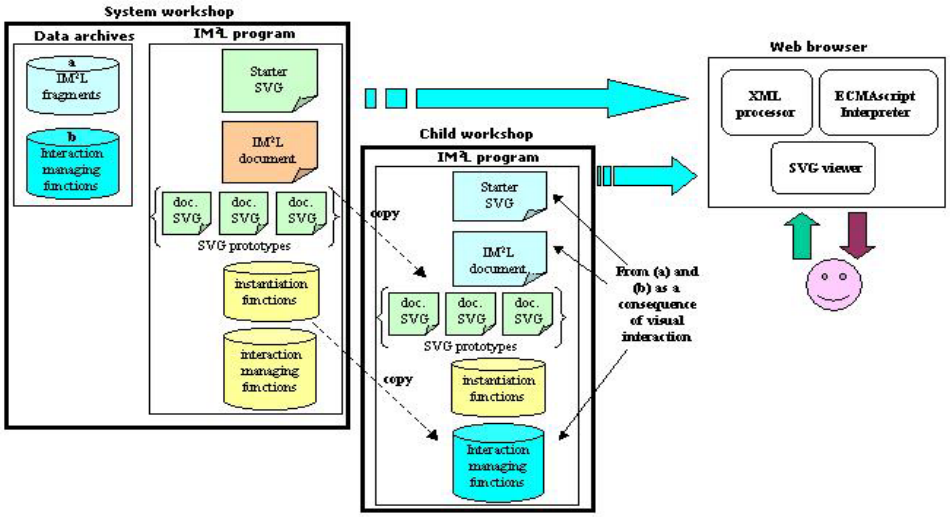

Fig. 6. The web-based architecture of a system workshop and the generation of a daughter workshop.

defined by the text included within the opening tag `<button>` and the closing tag `</button>`. As shown in Fig. 6, an IM^{2}L program includes some XML-based or XML-compliant (ECMAscript) documents and is interpreted by a web browser (see tick arrows), which coordinates the activities of a standard XML processor, an ECMAscript interpreter and an SVG viewer. SVG is the W3C standard for vector graphics [28].

In particular, an IM^{2}L program implementing a workshop includes (see Fig. 6):

a) a *Starter SVG*. This is the starter system linking the IM^{2}L document with its interpreter;
b) an *IM^{2}L document*, composed of one or more IM^{2}L fragments specifying the contents and the logical structure of the workshop and of the ves composing it;
c) the set of *SVG prototypes*, specifying the physical materialization of the ves and their topological relations;
d) the set of *instantiation functions*, which are used to instantiate the SVG prototypes with the information included in the IM^{2}L document, to compute their materialization features, such as geometry, color, appearance. The result of the instantiation is used by the viewer to establish the whole image on the screen;
e) the set of *interaction managing functions*, which implement the transformation rules of virtual entities defined in the design phase. They manage the interaction of the user with the workshop, possibly re-calculating the topological and geometrical features of virtual entities whenever their appearance or position must be modified or starting some computational activity in reaction to the user action.

Details about the initial state instantiation within the web browser of a SSW and how the interaction with it occurs can be found in [29]. Herein, we would like to underline the direct mapping existing between the design specification and the implementation.

The IM^{2}L document contains the description of the initial states of the ve*s* composing the SSW: actually, it specifies the vs_0 of the SSW independently of its materialization. Materialization details are specified in the SVG prototypes, and the instantiation functions permit the creation at runtime of the vs_0 of the SSW by combining the information present in the IM^{2}L document and in the SVG prototypes.

The interaction managing functions implement the transformation rules: the interaction of the user with the current state of the SSW means that a particular ve composing the SSW receives an input event and that a related function is called. Such a function may react by: 1) transforming the current state (cp_i) of the ve into another state (cp_{i+1}) so transforming the current state of SSW (vs_i) into another state (vs_{i+1}), being a composition of the cp*s* of the active ve*s*; 2) generating a new ve in its initial state cp_0, so transforming the current state of SSW (vs_i) into another state (vs_{i+1}) by adding a new cp to vs_i.

We will now discuss the generation of a daughter workshop while interacting with a system workshop. To this end, two special archives of data are exploited. These archives contain IM^{2}L fragments defining the virtual entities that the user will select to compose daughter workshops, and the interaction managing functions implementing the transformation rules of such virtual entities (see top left of Fig. 6). The IM^{2}L program implementing a daughter workshop is thus generated by the IM^{2}L program, implementing a system workshop in the following way:

(1) The archives containing the SVG prototypes and the instantiation functions are simply replicated. The underlying hypothesis is that only software engineers are in charge of preparing such kinds of information, since they are written using computer-oriented languages that are not easily manageable by end users and HCI experts.

(2) The IM^{2}L document is generated as a consequence of the selection of virtual entities from those available in suitable repositories and their combination into complex virtual entities, up to the whole workshop. The user interacts with the system workshop by direct manipulation, dragging virtual entities from repositories visualized on the screen to the working area (see Fig. 2); as a consequence, the IM^{2}L fragments defining the selected virtual entities are automatically searched for in the external archive of IM^{2}L fragments.

(3) The interaction managing functions of the daughter workshops are also searched for in the data archive as a consequence of the virtual entities selection: in fact, they implement the activities to be performed when the user interacts with the virtual entities in the daughter workshop, so links to them are present in the IM^{2}L fragments defining the virtual entities. All interaction managing functions linked to the IM^{2}L fragments, chosen during the daughter workshop composition, are therefore included in the archive of interaction managing functions.

(4) The starter SVG is generated in the saving phase of the daughter workshop to permit the initial loading of the IM^{2}L document into the browser.

7. Related work

Many researchers in HCI and Software Engineering (SE) have studied and tried to solve the communication gap existing between users and software designers. As already said, such stakeholders possess distinct types of knowledge and follow different approaches and reasoning strategies for modelling, performing and documenting the tasks to be carried out in a given application domain: users do not understand designers' jargon and designers in general do not understand the user's specific domain jargon [15]. It is important to notice that HCI experts, often called upon to represent user views in the design, own a specific knowledge themselves, which is not that of the users or of software designers [30]. Only users are able to read the screen thanks to their tacit knowledge, and understand what is misleading or difficult to interpret for them, but they are not able to think as HCI experts or propose adequate HCI solutions [31]. Neither users nor HCI experts can evaluate the technical consequences of their proposals nor the influence of the adopted technologies, i.e. they are not able to think as software engineers [1]. The loop is closed by software engineers, who know the technology, but in turn have difficulties in thinking as users or HCI experts. As emphasized by activity theory studies, the implicit knowledge is embedded in the tools and notations users adopt: tools and notations that depend on the context of activity and work organization [32].

SE and HCI experts are aware of the gaps existing among them and of the need to communicate and share their different points of view during the VIS design process. Lauesen [30] proposes the *virtual window method*, an early graphical realization of the data presentation to bridge the gap between software engineers and HCI experts. Folmer et al. [33] propose *bridging patterns*, which describe a usability design solution and consist of a user interface part and an architecture/implementation part. Borchers [34] recognizes the need to capture the knowledge of end users, together with HCI and SE expertise, by forging a lingua franca that makes the design experience understandable by domain experts, HCI experts and software engineers. He proposes a *pattern framework* in which three design pattern languages are used to bridge the gaps: the first describes the application domain, the second leads from domain and task analysis to interaction design and the third proposes software design solutions for the interactive system. The languages are formally structured in a hypertext graph notation, which underlies the definition and organization of the three pattern languages. Following this view, our approach to system modelling and design is based on the definition and use of at least two visual languages, one devoted to end users and HCI experts and the other to software engineers, each one permitting to specify the process of user-system interaction from a different point of view. The languages are not independent because they link the user views and jargons to the software engineer views and jargons, thus bridging the communication gap.

Sketching and modeling are integral features of the design process, critical for both the generation of ideas, and the communication of concepts to others for discussion and evaluation, particularly in the context of human-centered design. For this reason, a lot of interest has recently been devoted to *Rapid Prototyping* (RP) techniques, although in fact

RP has been a research topic of HCI researchers for more than 20 years [35]. Rapid prototyping is extensively used in Computer-Aided Design (CAD) and refers to a class of technologies that can automatically construct physical models from CAD data.

In the software engineering field, rapid prototyping offers a means to explore the essential features of a proposed system [36], [37], promoting early experimentation with alternative design choices and allowing engineers to pursue different solutions without any efficiency concerns [38]. Today, while there have been advances in the tools used, user interface prototyping remains the most effective way to gather requirements, communicate concepts between developers and users and evaluate usability in a cost-effective manner. RP is useful in software engineering to show the developed prototypes to the customers, but professional software tools are required to develop such prototypes, which are not suitable for use by end users. Each software environment developed according to the SSW methodology actually adopts RP techniques, but it is designed and developed to be suitable for end users who are professional people, but not experts in computer science.

In cooperative prototyping [39], prototyping is viewed as a cooperative activity between users and designers. Prototypes are developed by software engineers, then discussed with users, and possibly experienced by them in work-like situations. Prototype modifications may be made immediately by direct manipulation, also by users, during each session of participatory design. However, in this approach prototypes just represent an interactive digital evolution of paper-based mockups: real systems are then re-programmed and all modifications require a heavy programming effort, and are postponed and made by designers after each session.

In the SSW methodology, prototyping is also viewed as a cooperative incremental activity; in which the stakeholders participate in the actual development of the final system. Each stakeholder operates on prototypes according to their own view, through the use of SSWs. Moreover, end users are allowed to tailor their own SSW at run time.

Adaptation is currently a fundamental requirement in software systems. Adaptation is understood in the SSW methodology as the result of customization and tailoring of the workshops, both characterized by the explicit activity of the user; in literature the users' possibility to adapt the system to their needs is called adaptability [40], while adaptivity is the system's capability to automatically adapt itself to each individual end user [4][41]. In a workshop network, two different user populations participate to adapt a workshop: the domain experts in the design team, which use system workshops at design level, and end users, which use application workshops at use level. The term "customization" denotes the activity performed by the design team to develop application workshops for a specific community of users and the term "tailoring" denotes the activity performed by end users to adapt the system at run time to their current activity and context of work.

The SSW methodology emphasizes the need to develop different software environments for end users working in the same domain with different roles. Similarly, DAISY (Design Aid for Intelligent Support System) is a design methodology for building decision support systems in complex, experience-centered domains [42]. It provides a

technique for identifying the specialized needs of end users within a specific range of domain experience. In this sense, DAISY is a design methodology supporting the development of customized systems. In other domains, systems can have multiple end users with multiple roles. As an example, DIGBE (Dynamic Interaction Generation for Building Environments) is a system that creates end-user interfaces adapted to the multiple end users with different roles that collaborate to the management of a building control system [43]. Unlike software shaping workshops, software environments created using DIGBE are adaptive systems.

Software technology has advanced to the point where we can build tools end users can adopt to design systems by interacting with icons and menus in graphical micro worlds. Several researchers working on EUD have capitalized on this, and described technologies for component-based design environments, libraries of patterns, and templates (e.g. [11]). There are various proposed design environments that do not require users to program per se; instead, they design by instructing the machine to learn from examples [44]. From this perspective, system workshops devoted to domain experts permit the creation of programs just by visually composing virtual entities selected from repositories, as described in our case study in the mechanical engineering field.

8. Conclusions and discussion

The paper describes a design methodology for customized and tailorable visual interactive systems, that exploits the artisan's workshop metaphor. The emphasis on the proposed conceptual model and its relationship with the formal design are the novelties of this paper, which capitalizes on work previously performed by some of the authors (e.g., [15][24]). The objective of our work is to generate interactive systems that better fit users' needs and expectations. For this reason, our approach takes into account various problematic aspects of the Human-Computer Interaction process.

The users involved in the case study described in the paper understood and appreciated the novel approach of being involved in collaborative design processes, through which they can have a more active role than that of simple consumers of new technologies. The chance of having software environments that they can easily adapt to their needs is a new perspective that excited them very much. The results we have obtained so far, by observing people using the developed prototypes and collecting their comments and observations, are a clear sign that this approach may also determine an increase in end user productivity and performance, as well as more pleasure and fun in their overall experience with new technology.

The implementation architecture, supporting the recursive generation of SSWs as web applications, is also a novel proposal in the field.

Acknowledgements

The authors wish to thank G. Fresta for his contribution to the implementation of the prototypes. We are grateful to ETA Consulting in Brescia (and particularly to S. Biazzi)

for collaboration in the case study. We also wish to thank Ms M.V. Pragnell, B.A. for her help in correcting the English. This work was partially supported by the Italian MIUR and by EU and Regione Puglia under grant DIPIS.

References

1. D.J. Majhew, *Principles and Guideline in Software User Interface Design* (Prentice Hall, Englewood Cliffs, NJ, 1992).
2. P. Mussio, E-Documents as tools for the humanized management of community knowledge, Keynote Address in *Proc. ISD 2003*, Melbourne, AUS, 2003.
3. E. Folmer, M. van Welie, and J. Bosch, Bridging patterns: An approach to bridge gaps between SE and HCI, *Journal of Information and Software Technology* **48**(2) (2005), 69-89.
4. A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human Computer Interaction* (Prentice Hall, London, UK, 1998).
5. M. Polanyi, *The Tacit Dimension* (Rouledge & Kegan Paul, London, UK, 1967).
6. J.M. Carroll, and M.B. Rosson, Deliberated Evolution: Stalking the View Matcher in design space. *Human-Computer Interaction* **6**(3 and 4) (1992) 281-318.
7. G. Bourguin, A. Derycke, and J.C. Tarby, Beyond the Interface: Co-evolution inside Interactive Systems - A Proposal Founded on Activity Theory, in *Proc. IHM-HCI 2001*.
8. M.F. Costabile, D. Fogli, A. Marcante, and A. Piccinno, Supporting Interaction and Co-evolution of Users and Systems, in *Proc. AVI 2006*, Venezia, Italy, 2006, pp. 143-150.
9. J. Nielsen, *Usability Engineering* (Academic Press, San Diego, CA, 1993).
10. V. Rajlich, Changing the paradigm of software engineering, *Communications of ACM* **49**(8) (2006) 67-70.
11. A.I. Mørch, and N.D. Mehandjiev, Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units, *Computer Supported Cooperative Work*, **9**(1) (2000) 75-100.
12. H. Liberman, F. Paternò, and V. Wulf (Eds.), *End User Development* (Springer, Dordrecht, The Netherlands, 2006).
13. A. Sutcliffe, and N. Mehandjiev (Guest Editors), End-User Development, *Communications of the ACM* **47**(9) (2004) 31-32.
14. M.F. Costabile, D. Fogli, R. Lanzilotti, P. Mussio, and A. Piccinno, Supporting Work Practice through End User Development Environments, *Journal of Organizational and End User Computing* **18**(4) (2006) 43-65.
15. M.F. Costabile, D. Fogli, P. Mussio, and A. Piccinno, End-User Development: the Software Shaping Workshop Approach, in *End User Development*, eds. H. Liberman, F. Paternò, and V. Wulf, (Springer, Dordrecht, The Netherlands, 2006), pp. 183-205.
16. H. Rittel, Second-Generation Design Methods, in *Developments in Design Methodology*, ed. N. Cross, (John Wiley & Sons, New York, 1984), pp. 317-327.
17. J.C. Brancheau, and C.V. Brown, The Management of End-User Computing: Status and Directions, *ACM Computing Surveys* **25**(4) (1993) 437-482.
18. G. Fischer, Beyond 'Couch Potatoes': From Consumers to Designers and Active Contributors, *FirstMonday* (Peer-Reviewd Journal on the Internet), http://firstmonday.org/issues/issue7_12/fischer/.
19. M. Balaban, E. Barzilay, and M. Elhadad, Abstraction as a Means for End-User Computing in Creative Applications, *IEEE Trans. on Systems, Man, and Cybernetics - Part A*, **32**(6) (2002) 640-653.
20. EUD-Net Thematic Network, http://giove.cnuce.cnr.it/eud-net.htm.

21. G. Fischer, and E. Giaccardi, Meta-Design: A Framework for the Future of End-User Development, in *End User Development*, eds. H. Liberman, F. Paternò, and V. Wulf, (Springer, Dordrecht, The Netherlands, 2006), pp. 427-457.
22. D. Schuler, and A. Namioka, Preface, *Participatory Design, Principles and Practice*, (Lawrence Erlbaum Ass. Inc., Hillsday vii, NJ, 1993).
23. J. Preece, *Human-Computer Interaction* (Addison-Wesley Longman Ltd, Essex, UK, 1994).
24. P. Bottoni, M.F. Costabile, and P. Mussio, Specification and Dialogue Control of Visual Interaction through Visual Rewriting Systems, *ACM TOPLAS*. **21**(6) (1999) 1077-1136.
25. D. Fogli, A. Marcante, P. Mussio, L. Parasiliti Provenza, and A. Piccinno, A., Multi-facet Design of Interactive Systems through Visual Languages, in *Visual Languages for Interactive Computing: Definitions and Formalization*, ed. F. Ferri, (Idea Group Inc. Publication, in print).
26. B.A. Myers, and B.Vander Zanden, Environment for rapidly creating interactive design tools, *The Visual Computer; International Journal of Computer Graphics* **8**(2) (1992) 94-116.
27. D. Harel, On visual formalisms, *Communications of the ACM*, **31**(5) (1988) 514-530.
28. W3C: Scalable Vector Graphics (SVG), http: //www.w3.org/Graphics/SVG/.
29. D. Fogli, G. Fresta, A.. Marcante, and P. Mussio, IM^{2}L: A User Interface Description Language Supporting Electronic Annotation, in *Proc. Workshop on Developing User Interface with XML: Advances on User Interface Description Languages*, Gallipoli (LE), Italy, 2004, pp. 135-142.
30. S. Lauesen, *User Interface design - A software engineering perspective* (Addison-Wesley, 2005).
31. D.A. Norman, Human-centered design considered harmful. *Interactions* **12**(4) (2005) 14-19.
32. K. Kuutti, Activity Theory as a Potential Framework for Human-Computer Interaction, in *Context and Consciousness: Activity Theory and Human Computer Interaction*, ed. B. Nardi, (MIT Press, Cambridge, MA, 1995), pp. 17-44.
33. E. Folmer, M. van Welie, and J. Bosch, Bridging patterns: An approach to bridge gaps between SE and HCI, *Journal of Information and Software Technology*, **48**(2) (2005) 69-89.
34. J. Borchers, *A pattern approach to interactive design* (John Wiley & Sons Ltd., Chichester, UK, 2001).
35. M. Helendar (Ed.), *Handbook of Human Computer Interaction* (Elsevier Press, Amsterdam, North-Holland, 1987).
36. W. Hasselbring, Programming Languages and Systems for Prototyping Concurrent Applications, *ACM Computing Surveys* **32**(1) (2000) 43–79.
37. Luqi, Computer Aided System Prototyping, in *Proc. 1st Int'l Workshop on Rapid System Prototyping*, Los Alamitos, CA, USA, 1992, pp. 50-57.
38. R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven (Eds.), *Approaches to Prototyping* (Springer-Verlag, New York, NY, USA, 1984).
39. S. Bødker, and K. Grønbæk, Design in action: From prototyping by demonstration to cooperative prototyping, in *Design at work: Cooperative design of computer systems*, eds. J. Greenbaum, and M. Kyng (Lawrence Erlbaum Associates, 1991), pp. 197-218.
40. M. Krogsæter, R. Oppermann and C. G. Thomas, A User Interface Integrating Adaptability and Adaptivity, in *Adaptive User Support. Ergonomic Design of Manually and Automatically Adaptable Software*, ed. R. Oppermann, (Lawrence Erlbaum Associates, Hills dale, NJ, 1994), pp. 97-125.
41. L. Findlater, and J. McGrenere, A comparison of static, adaptive, and adaptable menus, in *Proc. of ACM CHI 2004*, Vienna, Austria, 2004, pp. 89-96.
42. C.B. Brodie, and C.C. Hayes, DAISY: A Decision Support Design Methodology for Complex, Experience-Centered Domains, *IEEE Trans. on Systems, Man, and Cybernetics - Part A* **32**(1) (2002) 50-71.

43. R.R. Penner, and E.S. Steinmetz, Model-Based Automation of the Design of User Interfaces to Digital Control Systems, *IEEE Trans. on Systems, Man, and Cybernetics - Part A* **32**(1) (2002) 41-49.
44. H. Lieberman, *Your Wish is My Command: Programming by Example* (Morgan Kaufman, San Francisco, CA, 2001).