Towards a Formal Notion of Interaction Pattern

Paolo Bottoni
Univ. La Sapienza (Rome)
bottoni@di.uniroma1.it

Esther Guerra
Univ. Carlos III (Madrid)
eguerra@inf.uc3m.es

Juan de Lara
Univ. Autónoma (Madrid)
Juan.deLara@uam.es

***Abstract*—While interaction patterns are becoming widespread in the field of interface design, their definitions do not enjoy a common standard yet, as is for software patterns. Moreover, patterns are developed for diverse design aspects, reflecting the complexity of the field. As a consequence, research on formalization of interaction patterns is not developed, and few attempts have been made to extend techniques developed for design pattern formalization. We show here how an extension to our recent approach to pattern formalization can be usefully employed to formalize some classes of interaction patterns, to express relations among them, and to detect conflicts.**

I. Introduction

Interaction patterns capitalize on experience on all different aspects of interaction design, and pattern collections are available in books [13], [14] and dedicated websites[1].

The movement towards patterns in HCI started under the influence of their success in Software Engineering (SE) [7], but had to face the specific problem of the distinction between the designer's view of patterns (as typical forms of collaborations among software components) and the user's view (where users are interested in reusing their experience from familiar widgets, layouts or navigation strategies) [6]. As a consequence, while the components of a pattern presentation are usually the same (as an example, see the Pattern Language Markup Language[2] for a comprehensive list of elements), there is no common notation to specify the solution, which is in most cases presented through examples and explanatory text, leaving it to the developer to code its details. Hence, it becomes hard to answer questions like: "Is **X** a new pattern or just a variation of **Y**, or even **Z** in disguise"? "Can I use **X** and **Y** together?" "Does the use of **X** depend on using **Y** in the same interaction?"

As HCI patterns involve combinations of requests on layout, individual or coordinated behaviours, or the structure of the domain model, presenting them only through examples makes it hard to separate essential aspects from features of the application domain. The compound of all these characteristics hinders the definition of a "real" pattern language, not restricted to simple pattern naming, but in which to express pattern composition, subtyping, dependency and conflict, so as to support pattern-based design.

In this paper we extend our algebraic formalization of a general notion of pattern [2] to found a notion of HCI pattern languages. To this end, we define mappings between components of an abstract User Interface specification (based on the UsiXML meta-model[3]) and the roles played by these components, thus developing methods to check whether an implementation is an instantiation of a pattern, to construct interface parts from specialisations of patterns, and to reason on pattern compatibility. We also extend the theory in [2] to describe relations between patterns, in particular pattern subtyping, and to model composition of HCI patterns, as well as conflicts or dependencies between them.

II. Related Work

Literature on HCI patterns is expanding to cover different aspects. For example, studies on usability have met architectural patterns to include interaction mechanisms from the start of the design process [8]. Folmer *et al.* relate architectural choices and usability patterns, through usability requirements which might have an impact on the architecture [5]. These are not expressed in terms of classes and relations, but define sets of problems the architecture has to solve. In this line, *bridging patterns* provide information on how to implement usability patterns [6].

Borchers [1] gives a notion of pattern language as a directed acyclic graph, where nodes are patterns and edges describe references from a pattern to another. However, the description of individual patterns does not rely on a formal characterisation, and the existence of pattern relations must be explicitly stated and cannot be derived from their analysis.

An abstract view of the components of an interactive system is at the core of the UsiXML proposal, combining approaches to model-driven platform-independent UI design [9] and abstract notions of interface objects [3]. In [11], the authors present a methodology exploiting abstract interaction objects to derive interaction patterns from analysis of domain and task models (e.g. patterns for handling entities or drawing associations between them) relating them to specific interaction and presentation techniques. The templates are defined in a semi-formal way but do not support the definition of relations among patterns.

[1]http://quince.infragistics.com, http://ui-patterns.com, http://www.welie.com/patterns
[2]http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html
[3]http://www.usixml.org/

III. Formal Model of Interaction Patterns

A pattern expresses a collaboration of elements playing specific roles to provide experimented solutions to recurring problems. In HCI patterns, collaborations can be implicit and simply recognised by the users, and roles can be played by any element in the interaction space. Hence, we separate the definition of a *pattern vocabulary* introducing the roles, from that of possible role realisations. Moreover, we allow for different types of collaborations by specifying a pattern as a collection of synchronized diagrams, with a designated *structuring* diagram introducing the roles. Also, diagrams contain *variability regions* constraining the number of elements which can play the same role in any given realisation of the pattern. For example, the *Button Group* pattern only makes sense when there are 2 or more buttons to be presented together.

As the resulting notion of pattern is domain-independent, domain-specific concepts can be used to express roles and to specify the elements realising them. In our approach, diagrams result from the annotation of model elements, typed on the UsiXML meta-model, with roles from the HCI pattern vocabulary. In addition, patterns are equipped with constraints (invariants), expressing contextual conditions on the correct application of the pattern.

A. A Meta-model for Interaction

We adopt the UsiXML meta-model to represent the interaction domain and relate its elements to pattern roles through a specific correspondence layer. Its specification provides a collection of modeling entities for the abstract and concrete definition of interactive systems. With the UsiXML meta-model, an interactive system is composed of several models, and an abstract user interface is realized through concrete elements and is connected to domain objects and workflow descriptions. In addition, we identify a vocabulary of roles as instances of the class `PatternRole` defined in Fig. 1.

In order to keep the domain and vocabulary models independent, we adopt triple graphs [12], where a *correspondence graph* relates the *source* and *target* graphs specifying the two models. In our approach, triple graphs are typed by meta-model triples, such as the one in Fig. 1. This has the advantage that any meta-model for interaction could be used, without affecting the definition of the roles. Roles are given a name, and attributed with a list of labels defining their *Focus*. Different UsiXML classes play different role types as given by the role maps in the correspondence meta-model. We have used abbreviations for these maps: from left to right, we have *Presentation*, *Affordance*, *Layout*, *Action*, *Container* and *Element*.

B. Pattern Specification

In its simplest form, a pattern consists of one *root* structure with the mandatory part that any pattern realization must contain, and a number of *variable* parts or variability regions defining additional structures that can be replicated several times for each instance of the root [2]. We use symbolic graphs [10], where data nodes are replaced by sorted variables, with a formula constraining their values.

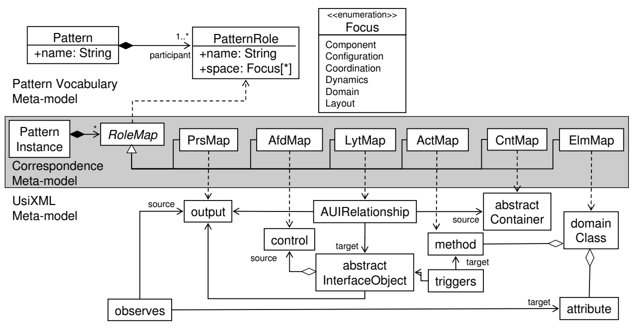

Figure 1. A fragment of the triple meta-model for the definition of Interaction Patterns.

Variable parts can be nested: a nested part can only be instantiated by adding structures to an instance of its parent. For each variable part, an *integer* variable is used in equations restricting the allowed number of its replicas. If the set of equations has no solution in the natural numbers, the pattern cannot be instantiated.

Def. 1 (Pattern): A pattern is a construct $VP = (P, root, Emb, name, var)$, where:

- $P = \{V_1, ..., V_n\}$ is a finite set of non-empty graphs, where each V_i is called *variable part*,
- $root \in P$ is a distinguished element of P,
- Emb is a set of morphisms $v_{i,j}\colon V_i \to V_j$ with $V_i, V_j \in P$, s.t. it spans a tree rooted in $root$ with all graphs $V_i \in P$ as nodes and the morphisms $v_{i,j} \in Emb$ as edges,
- $name\colon P \to L$ assigns each variable part a *name* from a set of variables L, of sort $\mathbb{N}$,
- $var \subseteq T_{AlgIEq}(name(P))$ is a set of equations governing the number of possible instantiations of the variable parts, using variables in $name(P) \subseteq L$.

The semantics of a pattern VP (written $SEM(VP)$) is given by the set of all valid expansions of its variability regions [2]. A model satisfies a variable pattern when some pattern expansion is found in the model. Fig. 2 shows the theoretical notation for pattern *ButtonGroup*, a compact notation we prefer to use, and an expansion where the *Action* variable part is replicated twice. The pattern contains a formula that enables expanding *Action* between 2 and 5 times.

As the definition of an HCI pattern involves several models, and in particular the abstract and concrete UI models, the problem of checking whether a model M of an interface satisfies a HCI pattern VP has to take care of this fact. In particular, M could specify only some components, for example providing only an abstract UI model, leaving the

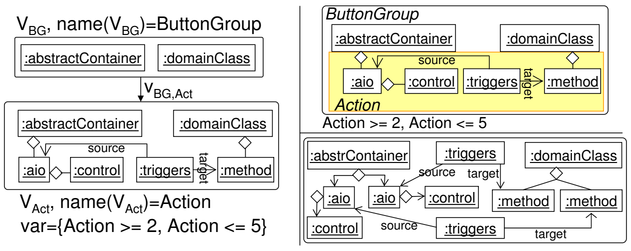

Figure 2. *ButtonGroup* in theoretical (left) and compact forms (top right), expansion (low right).

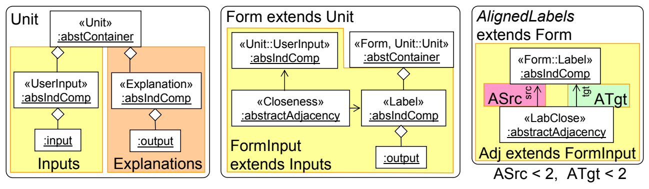

Figure 3. Unit (left), Form⊑Unit (center), AlignedLabels⊑Form (right).

choice of concrete widgets to developers. Hence, we will consider the expansions of VP only with respect to the UsiXML models used in M.

Using triple graphs as objects in the set P, elements in the model are annotated with their roles in the pattern. The source graph is given by a model in a domain-specific language (e.g. UsiXML), while the target contains nodes with the roles the elements can take. The assignment of roles to elements is made through the correspondence graph. As our formalization is given categorically, all definitions remain the same when replacing graphs by triple graphs. We do not explicitly show triple graphs, but use a compact notation similar to stereotypes, like the one in Fig. 3. As the definition of an HCI pattern may extend over several diagrams, we introduce synchronisation graphs to specify which elements in every variable part of a pattern correspond to each other and should be synchronized [2].

Patterns may also include conditions for their correct application, expressed as graph constraints [4]. A *pattern with invariants* is a pattern together with sets $PC(V_i)$ of *pattern constraints* $V_i \to X \to C_j$. An atomic constraint has one *premise* graph X (related to the variable part V_i it constrains) and a set of *consequence* graphs $C(X) = \{X \to C_j\}_{j \in J}$. If the premise graph X is found in a model, then some of the consequence graphs C_j have to be found as well. More complex constraints can be formed by using boolean formulae over atomic constraints. In particular, one can require that no instance of X be found.

IV. A Pattern Language

We extend our formalisation to handle pattern *subtyping*, *conflict* and *composition* to provide an effective basis for the construction of a HCI pattern language.

A. Subtyping

We start by identifying a *Unit* as the fundamental brick in the construction of a HCI pattern. A unit is formed by a *container* where some *individual components* contain *output* messages providing explanations to the user and some others offer facets for user *input*. The left of Fig. 3 presents a unit as a pattern with role realisations given via abstract elements from the *auiModel*.

A form is a unit presenting labels adjacent to the user input. Hence, we define *Form* as a subtype of *Unit*, noted $Form \sqsubseteq Unit$, by adding new elements and describing their relation to the elements of *Unit* as in Fig. 3 (center). Subtyping can add new elements to a pattern, possibly introducing further constraints, or adding new variable parts as children of the root or of other existing variable parts. However, the subtype cannot relax the constraints on the parent type, nor can it introduce intermediate regions between the root and the original regions. When extending a pattern, we only show in the child those elements of the parent pattern needed for the extension (but as in OO programming, all elements of the parent are incorporated into the child). Definition 2 formalises this idea.

Def. 2 (Extension): Given two patterns VP and VP', an injective morphism $Ext\colon VP \to VP'$ is defined as the tuple $Ext = (E = (E^V, E^E), f)$, where E is an injective morphism on trees that preserves the structure of the trees Emb and Emb' with:

- $E^V : P \to P'$ s.t. $E^V(root) = root'$,
- $E^E : Emb \to Emb'$ s.t. $E^E(v_{ij} : V_i \to V_j) = E^V(V_i) \to E^V(V_j) \in Emb'$,

and $f = \{f_i : V_i \to E^V(V_i) \mid V_i \in P\}$ is a set of injective (triple) graph morphisms s.t. the square (1) in the figure below commutes, and $\forall V_i \to X \to C_j \in PC(V_i)$, $\exists E^V(V_i) \to X' \to C'_j \in PC(E^V(V_i))$ s.t. squares (2) and (3) in the figure below are pushouts and $|C(X)| = |C(X')|$. Regarding the set of equations var, we demand $E^V(var) \subseteq var'$ and that no formula of $var' \setminus E^V(var)$ contains variables in $E^V(P)$.

$$\begin{tikzcd}
V_i \arrow[r, "f_i"] \arrow[d, "v_{ij}"'] \arrow[dr, phantom, "(1)"] & E^V(V_i) \arrow[d, "E^E(v_{ij})"] \\
V_j \arrow[r, "f_j"] & E^V(V_j)
\end{tikzcd}
\qquad
\begin{tikzcd}
V_i \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & X \arrow[r] \arrow[d] \arrow[dr, phantom, "(3)"] & C_j \arrow[d] \\
E^V(V_i) \arrow[r] & X' \arrow[r] & C'_j
\end{tikzcd}$$

Given patterns VP and VP', if $\exists Ext\colon VP \to VP'$ s.t. Exp preserves role names and focus we say that VP' extends VP, and we write it $VP' \sqsubseteq VP$.

The requirement of f for constraints of extended parts is that they should be provided exactly with the new elements added to the variable part they constrain (hence the pushouts) and they should not add new consequence graphs. However, VP' can add new premises X_k. The condition on the formula states that VP' cannot modify the formulae of VP, but can add equations involving new variable parts. In

Fig. 3, if a variable part V_i of VP is extended by $E^V(V_i)$ of VP', we label the extended part as "$name'(E^V(V_i))$ extends $name(V_i)$".

Our set-based semantics for patterns enables the usual replaceability of supertypes by subtypes as subsetting of the respective expansions, as stated in Theorem 1, whose proof is immediate.

Th. 1 (Subtyping): Given patterns VP and VP', if $VP' \sqsubseteq VP$ then $SEM(VP') \subseteq SEM(VP)$.

Specific types of *Form* require some forms of alignment of labels with inputs and between themselves. The Quince pattern repository distinguishes between left, right and top aligned labels. We introduce the abstract $AlignedLabels \sqsubseteq Form$ pattern to the right of Fig. 3, by adding a role for adjacency, with each label adjacent to one or two other ones.

Then, all of *Left Aligned Labels*, *Right Aligned Labels* and *Top Aligned Labels* are defined as subtypes of *AlignedLabels* by adding a set of suitable constraints for each specific type of alignment. For example, the left of Fig. 4 shows the constraints for left alignment. The pattern contains three constraints $X_i \to C_i$, presented as overlapping graphs. The first constraint states that if the user input and the label are reified by cio elements (X_1) those should be horizontally aligned (`grphAlgn` indicates an instance of *graphicalAlignment*). The second constraint states that if the label has a closeness adjacency with another label (of an instantiation of another *AlignedLabel*), then the cio reifications of both should be vertically aligned. Finally, the last constraint states that if the output is reified by the concrete *outputText* element, this should be aligned to the left.

B. Conflicts

When certain roles in different patterns are identified with each other, conflicts may arise either between the constraints of the pattern, or caused by incompatibilities with the integrity constraints of the domain specific language. The way to proceed is to encode the meta-model constraints using graph constraints, and then detect incompatibilities statically through compositions. For example, on the right of Fig. 4 a negative constraint (NAC) states that two elements can only share one type of *graphicalAlignment* at most. Hence, all the *AlignedLabel* specialisations are in conflict with each other. Indeed, if two pattern instantiations share a common `UserInput`, $X_1 \to C_1$ would demand two *graphicalAlignments*, in contradiction with the global constraint. These conflicts can be computed statically by performing pattern compositions, described next.

C. Pattern Composition

To compose two patterns, one glues their roots via their pushout through elements selected to be identified, yielding the root of a new composite pattern. The process is repeated for the elements to be identified in the variable parts. Merged variable parts receive the same name, the original equations are united (after renaming), and most restrictive ones subsume the others. As an example, in Fig. 5 the Quince patterns *Command Area* and *Clear Entry Points* are composed. The first pattern groups commands together (modelled by variable part *Commands*) into a unified area of the interface (role *CommandArea*). The second pattern provides a set of entry points (region *Entries*) into an application or Web site, based on their most common tasks or destinations. The diagram to the left shows the composition scheme. In the resulting pattern, the root elements with roles *Main* and *HomePage* are identified (graph I_r), as well as *Command* and *EntryPoint* (and linked object `control`) in the variable parts (graph I_{VP}). The composed pattern is built by the pushouts of the roots and the variable parts, where the embedding u of the resulting root is given by the universal pushout property. The resulting pattern groups commands to entry points into a common area.

$$\begin{tikzcd}
& I_r \arrow[dl] \arrow[dr] & \\
\text{CommandArea} \arrow[dr] \arrow[ddd] & \text{P.O.} & \text{ClearEntryPoints} \arrow[dl] \arrow[ddd] \\
& \text{EntryPointArea} \arrow[d, "u"] & \\
= & & = \\
\text{Commands} \arrow[r] & \text{CEntries} & \text{Entries} \arrow[l] \\
& \text{P.O.} & \\
& I_{VP} \arrow[uul] \arrow[uur] &
\end{tikzcd}$$

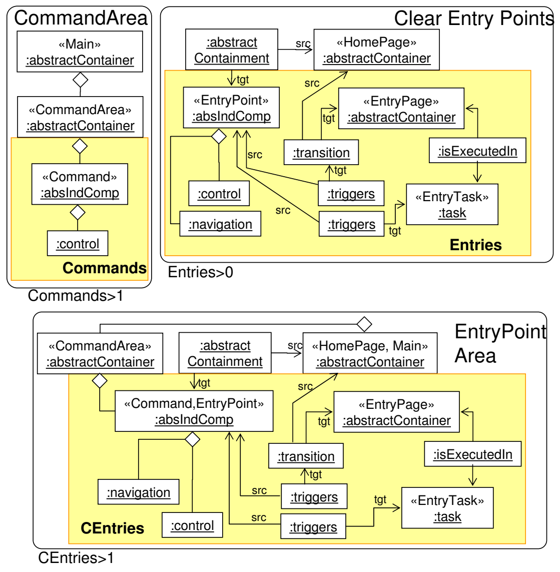

Figure 5. Composing patterns.

V. Conclusions and Future Work

The theoretical foundation of a notion of HCI pattern language is challenged by the heterogeneity of the involved aspects, from classical SE concerns to cognitive issues. Currently, the definition of pattern languages is based on structured textual descriptions of motivations, contexts and solutions, and exemplar realisations. Hence, relations between patterns can only be explicitly posed, typically in terms of dependencies and conflicts, but they cannot be properly identified, nor can it be determined if some implementation is a realisation of a known pattern.

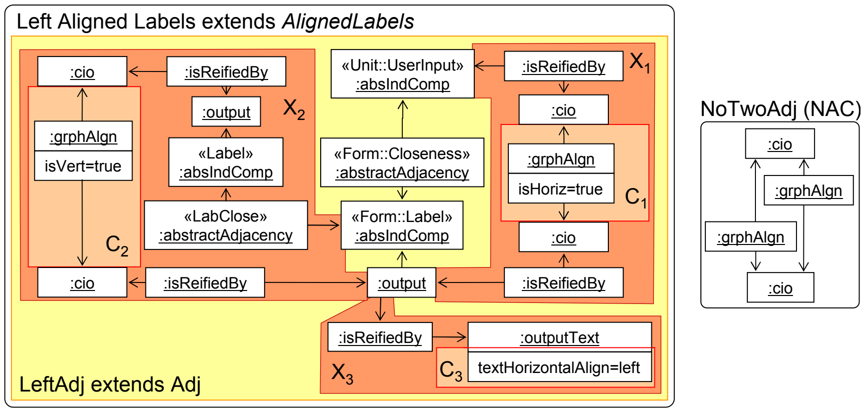

Figure 4. Constraints for *LeftAlignedLabels* (left). Global Negative Condition (right).

A recently proposed formal definition of pattern, based on triple graphs, has been used here to describe the solution component of a HCI pattern, with reference to its several aspects and different levels of abstraction. In particular, the identification of relevant roles in a pattern, from the point of view of the domain, presentation and dynamics aspects of interaction definition, provides a basis for determining the existence of relations between patterns, in particular subtyping, conflicts and decomposition.

We plan to test this approach on complete existing collections of patterns, possibly identifying common abstractions for families of patterns and exploring the limitations of the current proposal. In particular, we envisage that imposing the structure of a directed acyclic graph, rather than of a tree, on the set of morphisms between variable parts will conquer more patterns to formalisation.

Acknowledgement

Work funded by the Spanish Ministry of Science and Innovation through mobility grants JC2009-00015 and PR2009-0019, project TIN2008-02081 (METEORIC) and the R&D programme of the Madrid Community, project S2009/TIC-1650 (e-Madrid).

References

[1] J. Borchers. *A Pattern Approach to Interaction Design*. Wiley, 2001.

[2] P. Bottoni, E. Guerra, and J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Inf. Soft. Technol.*, 52(8):821–844, 2010.

[3] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[4] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundam. Inform.*, 74(1):135–166, 2006.

[5] E. Folmer, J. van Gurp, and J. Bosch. A framework for capturing the relationship between usability and software architecture. *Software Process: Improvement and Practice*, 8(2):67–87, 2003.

[6] E. Folmer, M. van Welie, and J. Bosch. Bridging patterns: An approach to bridge gaps between SE and HCI. *Information & Software Technology*, 48(2):69–89, 2006.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[8] N. J. Juzgado, M. López, A. M. Moreno, and M. I. S. Segura. Improving software usability through architectural patterns. In *SE-HCI*, pages 12–19. IFIP, 2003.

[9] G. Mori, F. Paternò, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE TSE*, 30:507–520, 2004.

[10] F. Orejas. Attributed graph constraints. In *ICGT'08*, volume 5214 of *LNCS*, pages 274–288. Springer, 2008.

[11] C. Pribeanu and J. Vanderdonckt. A transformational approach for pattern-based design of user interfaces. In *ICAS*, pages 47–54. IEEE Computer Society, 2008.

[12] A. Schürr. Specification of graph translators with triple graph grammars. In *WG*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[13] B. Scott and T. Neil. *Designing Web Interfaces: Principles and Patterns for Rich Interactions*. O'Reilly, 2009.

[14] J. Tidwell. *Designing Interfaces*. O'Reilly, 2006.