Measuring Interaction Design

1st Author Name
Affiliation
City, Country
e-mail address

2nd Author Name
Affiliation
City, Country
e-mail address

ABSTRACT

Early prototyping of user interfaces is an established good practice in interactive system development. However, prototypes cover only some usage scenarios, and questions dealing with number of required steps, possible interaction paths or impact of possible user errors can be answered only for the specific scenarios and only after tedious manual inspection.

We present a tool (MIGtool) that transforms models of a user interface into a graph, upon which usage scenarios can be easily specified, and used to compute possible interaction paths. Metrics based on possible paths, with or without user errors, can be easily computed. For example, when applying them to compare mail applications, we show that Gmail has 3 times more shortest routes, has 2x more routes that include a single user error, has routes with 13% fewer steps, but optimal routes have the smallest probability to be chosen.

Author Keywords

Experimental; Evaluation; Statecharts: UML; Metrics; Testing.

ACM Classification Keywords

H.5.1 Information interfaces and presentation (e.g., HCI): Multimedia Information Systems.; H.5.2 Information interfaces and presentation (e.g., HCI): User Interfaces.

INTRODUCTION

We present an approach that allows a designer to assess interaction design (IxD) qualities such as efficiency, error-proneness and recovery from errors. Key importance is given to the ability to (a) understand how supportive a user interface (UI) is with respect to user efficiency; (b) understand how prone the UI is to user navigation errors; (c) understand how recoverable the UI is from those errors; and, (d) perform objective quantitative comparison of different designs to support construction and engineering of interactive systems. All this can be done before developing prototypes of the UI.

To explain our work we present a comparison of four web mail front ends. We have chosen this domain because it is easy to understand and yet several questions cannot be easily answered. We applied the same techniques also in other domains, such as HVAC and other embedded UIs.

Developing good UIs for web or mobile applications is a complex and expensive endeavour. One reason is the combination of devices, interaction modalities and workflows that need to be supported. Adopting Usage Centered Development (UCD) practices is effective, as is following established design principles [9]. In particular, one of the most effective technique is early prototyping to explore part of the five-dimensional fidelity prototyping space [18, 7]. It is particularly effective when paired with usability investigations based on user testing or heuristic evaluations [27, 28].

However, UCD requires prototypes which are usually developed with certain tasks in mind, and therefore are quite restricted in terms of depth, breadth, dynamics and data. Furthermore, in addition to the possible bias introduced by prototypes, usability results are always surrounded by a cloud of uncertainty, due to subjectivity introduced by participants and facilitators or by other contingency factors involved in the analysis. Thus, although a significant effort needs to be directed to develop and use prototypes, less than optimal results are obtained.

Even worse, several questions cannot be easily answered. Given one or more potential designs and some usage scenarios, interesting questions include: “How many different routes can be followed by the user to carry out the scenario?”, “Which are the shortest ones?”, “If a user makes a mistake, would he or she be able to recover?”, “How many steps would the recovery require?”. When designing and evaluating embedded UIs (such as when dealing with plane’s cockpits [4]), other relevant questions include “How would the above properties change if we add a certain a widget?”, or “... if we replace a widget with another?”. At the moment, these straightforward questions are quite complicated to answer. In fact, they require development of prototypes, inspecting them, manual tracking of which screens and widgets are used at which stage, and a systematic manual analysis.

This should not be the case, however, because answers to these questions can be automatically computed. In this way a system can provide important insights to a designer, and support assements of potential user flexibility, user efficiency, error proneness, ability to recover, compactness and consistency of a design.

Our approach is based on UML statechart models of UIs which are automatically processed to produce *interaction* *graphs*. These graphs are then used to specify interaction scenarios and to unfold the possible interaction paths (called *execution traces*) that are compatible with the specific scenarios being considered. Traces are fed to graph-theoretic computations which produce a dashboard with different results. Except for development of models and specification of the desired scenarios, which have to be done manually, the other steps are totally automatic; models of a design (such as the ones shown below) can be developed in a matter of a couple of hours.

Our contribution consists of (a) the idea of using a UML state machine model to specify interaction scenarios, (b) the development of a tool (MIGtool, Measuring Interaction Graphs Tool) that transforms models and scenario specifications into interaction paths, and (c) the definition of metrics that provide concise, precise and objective measures of a design. The examples presented below show that among four web mail applications, and with respect to a typical usage scenario, Gmail is the most efficient and flexible UI, with the best ability to recover from user errors, but only for users that possess a certain level of proficiency. In fact, Gmail has the largest number of shortest paths (when users are supposed to make 0 or 1 error); when users make 2 or more errors the number of paths drops significantly, reducing the error-proneness of the UI; on the best case, Gmail features also the shortest paths, requiring 13% fewer steps; however, the probability that a user hits an optimal path with Gmail is 10 times smaller than the best of the other applications, and the probability that a random walker hits a state that is not due to an error is 10% smaller than the best of the other applications. These values suggest that Gmail offers more ways to accomplish tasks included in the scenario, that comparably more of these ways do not involve extra steps, that they require fewer steps, and that it might be more difficult for novice users to exploit the most efficient ways. Further inspections show that some differences are due to the slightly different interaction structures adopted for uploading messages. If Gmail didn't exist yet, by using MIGtool its designers could obtain these answers well before developing prototypes and performing usability studies.

Other examples shown below show what is gained when a new feature is added to an embedded system.

BACKGROUND

The literature on using state-transition networks for specifying or analysing the behavior of UIs is vast. We conducted a systematic-style literature review, using Google Scholar and queries with combinations of these phrases: "user interface", "path analysis", "user trace", "interaction trace", "navigation", "markov chain", "markov model", "state transition", "statechart", "metric", "measure". For each query we analyzed title and abstract of the first 50 hits and appraised their relevance based on whether the paper discusses approaches for measuring user interfaces in the context of usability and on the basis of a state transition model. This resulted in 78 full-text papers that were later on re-analyzed against the same criterion, leading to several of papers mentioned below. Many papers deal with testing user interfaces, and problems related to generating execution traces as a means to assess coverage of a test suite [2, 6]. They were excluded from the review.

Usage of state-transition networks to model UI behavior in order to draw usability conclusions dates back at least to [25]. In it, Parnas claimed that several kinds of usability problems would not occur if the designer adopted a design framework where states and their transitions are made explicit.

In many cases *statecharts* are used, a generalization of finite state automata. Horrocks showed how statecharts can be used to model and specify the dynamics of typical desktop UIs [15]. While providing many interesting insights on how and why one should use statecharts to do so, this nice work does not address how such a specification could be *automatically* processed. This idea was later on expanded by Thimbleby [31]; statecharts are seen as crucial representations that allow a designer to fully appreciate how devices behave. The overall claim is that "If you don't understand the logic conveyed by a statechart model of a user interface, then you don't understand the behavior of the user interface".

WebML [8] is one of the most successful model-driven approaches to web development (UIs and backend systems), with industrial traction and a large number of publications. The language is based on state transitions and is targeted to automatic generation of data-intensive web applications. Many other similar approaches involve or are based on task or activity models [20, 26, 17, 10, 19]. A recent OMG standard, called Interaction Flow Modeling Language (IFML) [21], derives from WebML and focuses specifically on user interaction. IFML is a language for specifying the structure of a user interface and its behavior. It offers most of the abstractions that are available in statecharts, mixed with the ability to specify so-called "components" that are used to display and manipulate data (to display details of a item, to display or select lists of items, to input an item). None of these approaches focus directly on measures of usability.

A rather different route for the problem of generating UIs is followed in [12]. Authors assume that the UI to be generated is used to supervise and to monitor an underlying machine (*e.g.*, autopilot of a plane) which is modeled as a statechart. After assuming that the behavior of the UI can also be modeled as a statechart, they devised an algorithm that checks whether the two models are compatible, and that refines the UI model so that its states and transitions are minimized while still allowing a correct manipulation of the underlying machine. Application of such a technique leads to UIs that are correct by-construction.

In [4] the Interactive Cooperative Objects formalism is discussed, which is based on Petri nets, yet another state transition formalism. It is used to enrich the ARINC 661 specification of Cockpit Display System so that the semantics of widgets can be expressed and the behavior of a UI be formally analyzed.

Finite state representations have been used also as a conceptual framework for writing the code of widgets so that events and event handlers in the UI can more easily be conceived, developed and verified (*e.g.*, [3]).

A review of usability measuring practices [14] lists, under the headings *measures of efficiency/usage patterns*, number of keystrokes, mouse clicks, and visited objects as possible metrics. The notion of *deviation from optimal solution* is discussed only in the context of tools for route planning in 3D navigation. In this paper we provide our own definition of deviation, which applies to interaction with the UI; we provide also our notion of execution traces across states of the UI, and the length of these traces can be used as a measure of efficiency.

Lostness [30, 23] is one of the few metrics for measuring the degree to which users become lost in the information space, and therefore considers the notion of *deviation*. Defined specifically for hypertext systems, lostness is a user performance measure which is afuncion of the number of visited nodes, the number of different nodes that were visited nodes, and the number of required nodes. This measure of efficiency is usually applied to traces of actual users, and is argued to be suitable for hypertext systems because the predominant task is browsing information, rather than trying to achieve specific goals. It is suitable therefore whe some of the following three assumptions can be relaxed: that there is a task to complete, that there is a correct way to carry it out, that the purpose of the system is to support users to carry out their tasks.

A usability analysis method capable to analytically predict task completion times from a storyboard of the UI is CogTool [5]. CogTool relies on the ACT-R cognitive modeling engine, and allows a designer to setup a low-fidelity prototype of the UI. After deciding which interaction modality and which widgets are used to implement actions, the designer gets an estimation of how long an experienced user would spend on each step. CogTool takes care of adding extra "mental" steps, such as "think" steps, before certain patterns of provided steps, according to the cognitive theory underlying ACT-R. As a result, users of CogTool obtain the breakdown of the times required by each of the steps. Our method is weaker in terms of cognitive validity and in terms of precision of the output: it does not provide expected completion times. However, with our method one can analyze a large part of the UI, get information about possible problems in some areas, and only then devolve more resources in building storyboards and in making assumptions regarding widgets so that specific execution paths previously identified can be analyzed with CogTool. In a sense, the output of our method could be used to make informed decisions as to what to analyze with CogTool.

Markov models, *i.e.* directed graphs where edges leaving a vertex are associated to a probability distribution, were used in [32], as a means to perform usability analysis as early as possible, even before building prototypes of the UI. Vertices represent states of the UI and edges correspond to user actions (such as pressing a push-down button). Probabilities can be used as a model of user knowledge: equiprobable actions correspond to a knowledge-free user, whereas when some actions have a very low probability it means that for that user the action is unlikely to be executed. Simple mathematical operations on the transition matrix of the model give the probability that after *n* steps from a given initial state the UI is in a given state. With Markov models, by manipulating probabilities, the designer can plot the number of required steps as a function of how close the probabilities are to the designer's "perfect" knowledge. Examples discussed in the paper cover several devices, ranging from a simple torch (with 4 states), a microwave cooker (6 states), a mobile phone (152 states); those are all push-down devices with a fixed set of buttons. This is obviously not the case for UIs of information systems, where buttons may change screen by screen. This makes it more difficult to specify the transition matrix. Our approach is based on statecharts, a language that in practice is more powerful than Markov models, making it easier to specify the UI behavior, especially in cases where the set of buttons change over time. While our examples do not make use of probabilities, this is very easy to cope with (see the Discussion at the end of the paper for some of the benefits that doing this could bring). Similarly to [32], our approach could be used when conservative results that do not rely on psychological assumptions are sought.

A discussion of social network analysis metrics applied to interaction design is provided in [33]. Once more, a UI is modeled in terms of directed graphs (vertices are states and edges are actions), and various centrality metrics are used to draw conclusions that bear upon usability. For example, centrality measures (such as Sabidussi, eccentricity, betweenness) can be used to identify states that are good places to start from to get to other states. Other metrics, such as edge betweenness, can be used to identify actions that are important because most of the shortest paths go through these actions. The paper presents compelling examples of using this technique to identify shortcomings in the design of infusion pumps. In our work we automatically generate graphs from statechart models, and on some of them computed these metrics. We were not able to draw sensible conclusions from the values we obtained (for example from the models presented below). One possible explanation rests on the different types of models: in our case they reflect the variety and flexibility with which "buttons" can be used in modern web applications. For infusion pumps the UI is more constrained in how a task can be carried on, and this difference might reflect on the usefulness of those metrics.

In [13] several approaches to analyse streams of user events are discussed and compared. It is interesting to realize that this is, in a sense, the inverse problem of the one we tackle in this paper: we want to compute a subset of the possible streams of user events given a specification of the UI, rather than trying to abstract general properties from streams of events.

GENERATION OF INTERACTION TRACES

In this section we provide a description of how MIGtool computes execution traces. They are paths (*i.e.*, sequences of connected states of the model) that users can follow when performing a given scenario. The generation process encompasses the following steps: (1) automatic flattening of the statechart model; (2) manual definition of the interaction scenario; (3) automatic generation of execution traces; and (4) interactive analysis of results.

Processing models

MIGtool takes as input UML statecharts, which are a generalization of finite state automata (FSA), with an expressive language that includes hierarchic levels of abstraction, concurrent regions, states and pseudostates, guards, and an extended state notion based on an arbitrary underlying computational model. Harel, the inventor of statecharts, gives an interesting retrospective view of how they were invented, back in early eighties, and why they were appealing also to non experts [11]. For the sake of brevity, we refrain from describing them here in detail; the interested reader is referred to the UML standard [22] or some of the textbooks that deal with them, like [29].

By using statecharts, the behavior of UIs can be represented by associating states to screens and particular configurations of widgets, and transitions to actions performed by users or by the system itself. In the classification reported in [13], actions belong to the "abstract interaction events" category.

Because statechart models take advantage of abstraction features and a rich set of connecting pseudostates, they are not suitable to be directly processed to compute metrics. For this reason, MIGtool first *flattens* the model. Flattening is a process often used when statecharts have to be automatically processed [6], and it means to produce a FSA that is behaviorally equivalent to the original statechart, with no hierarchy between states and no concurrent regions. In most cases this leads to an exponential number of states and transitions in the FSA, but because the process is completely automatic and there is no need to manually inspect the resulting FSA, this aspect is in many cases not relevant. MIGtool produces an XML representation (graphML) of the resulting FSA, the *interaction graph*. It is a directed multigraph[1], potentially with cycles and loops, with edges that are labeled with the name of the corresponding action. In the following we will be using as synonymous the terms state/vertex, and action/edge.

Defining usage scenarios

Because in all but the most trivial interaction graphs there are cycles, the set of possible interaction traces is infinite. For this reason, scenarios need to be defined and used as constraints on the possible execution traces that can be generated. Users of MIGtool define interaction scenarios by specifying key steps (called *bridge sets*) that users are expected to go through; we call this process *grounding usage scenarios on models*. Each bridge set is specified by selecting a subset of the edges of the interaction graph. In general a specification includes an initial state and a sequence of bridge sets. For example, to specify a scenario for replying to an email message, one could select all the edges associated to the action `reply` (bridge set 1), followed by edges labeled with `typeBody` (bridge set 2), followed by edges labeled with `send` (bridge set 3). A well formed bridge set consists of 1 or more edges (an empty bridge set would make the scenario unviable). In this way scenarios with cycles can be formulated (*e.g.*, reply twice to two messages). A *stage* of a scenario comprises two consecutive bridge sets. To cope with multi edges, the interaction graph is *simplified*: all edges between a pair of vertices are merged into a single one, whose label includes the original ones.

[1]A directed multigraph is a graph such that there are 2 or more edges that have the same end points. Cycles are paths that include 2 or more occurrences of the same vertex. Loops are edges that start and end on the same vertex. "Geodesic path" is a synonymous term of "shortest path".

Searching traces

Quite expressive languages can be conceived for grounding scenarios (*e.g.*, regular expressions on sequences of action names). But such expressivity bonus needs to be balanced with computational tractability: even models with a dozen states might correspond to interaction graphs with several hundred states and several thousand edges, leading to an enormous number of possible traces to filter even for scenarios with just a few stages.

To cope with this we implemented a trace searching algorithm that processes each of the stages sequentially, starting from the initial initial state. Given a bridge set B_i, the algorithm does a breadth-first search of all the geodesic paths[2] that connect each of the ending vertices of edges in the bridge set B_i with some of the starting vertices of edges in bridge set B_{i+1}. If some of the starting vertices cannot be reached, then the bridge is dropped. MIGtool creates a new graph from the geodesic paths found for each stage, and then joins these graphs so that geodesic paths found for stage i are joined with those of $i + 1$. These global paths, connecting the initial state and the remaining bridges of the last bridge set of the scenario are called *execution traces*.

Notice that a scenario specifies only the desired occurrences of actions, not all the necessary ones. For example, if the model prescribes that in order to `view(message)` while reading another one, one has to `goBack` to the inbox first, a scenario specifying two consecutive `view(message)` would lead to geodesic paths that include also the `goBack` action, even if it is not specified in the scenario. It is the task of MIGtool to unfold cyles in the graph and search all the geodesic paths that connect the desired user actions.

Detour traces

The algorithm described so far finds (some of) the global paths connecting the initial state to one or more bridges for each of the specified stages. The globally shortest path is identified, together with other viable alternatives. These traces are called *detour order 0* traces (and states).

For each stage, up to a maximum detour order H, the search algorithm creates traces with detour order $k + 1$ by collecting states D_k with order 0 up to k, and by identifying the neighbours (N_k) of D_k (which are the states not included in D_k that are connected through an edge to some state in D_k). Edges connecting D_k with N_k represent deviations that users might follow, and geodesic paths between N_k and D_k constitute recovery paths that users might follow to complete the scenario from states in N_k. These deviations and recovery paths are joined and constitute the traces of order $k + 1$. It could happen that for some state in N_k there is no path leading to any

[2]"Geodesic" is synonymous with "shortest".

vertex in D_k: in such a case the deviation leads to a dead end that prevents the user to complete the scenario.

In general, MIGtool is used to process a model against a scenario and to generate execution traces of detour order = 0, ..., H. The time complexity of the search algorithm is $O(KM(E + V))$, where there are K bridge sets, the mean size of them is M, the interaction graph has V vertices and E edges. Therefore it scales pretty well with the size of the interaction graph and/or complexity of the scenarios. In practice for interaction graphs consisting of about 10000 edges and a dozen of bridge sets, on a low-cost PC it takes about 15 seconds to generate traces of order 0 to 3.

COMPARING APPLICATIONS

In this section we describe some examples based on well known web mail clients, namely Gmail, Horde, SquirrelMail and Roundcube. We chose these examples because they are very well known, and therefore are easy to describe and understand. And yet, despite email being a very well understood domain, the kind of questions that can be posed and the answers that are found provide an insight on some of the usability properties of these applications.

Models and scenarios

Models of the four applications have been manually developed using UML state machines. In order to support a fair comparison, all four models cover the same set of use cases: listing the content of the inbox, reading a message or conversation, replying to a message, composing and sending a new message.

Figure 1 shows part of the model of Gmail. At some point, the user might be viewing all the conversations of the inbox (state `viewingConversations`); available actions include moving to the next or previuous block of conversations, refreshing the list, or opening a specific conversation (transition `open(conversation)`). This transition (which is assumed to occur when the user clicks on one of the visible conversations) leads to the state `viewingAConversation`, where the behavior of the system is defined by a more detailed state machine. By default the user is viewing a conversation, but by performing the `reply` action the UI moves to a state called `replying`, where the body of the message can be typed, the subject can be changed, or another addressee can be added.

Notice that this behavior "happens" in one of the two concurrent regions specified by this model. In parallel to this, the user can either be reading messages (state `reading`) or may be composing a new message (state `composing`). In the latter case the user can independently add recipients, attachments, write the body or subject of the message; and send or cancel it.

The model that we show here is part of what we used in the examples reported below, and that is only a part of what the real Gmail application supports. The actual model that we used consists of 24 states, 12 pseudo states, 12 regions, and 61 transitions.

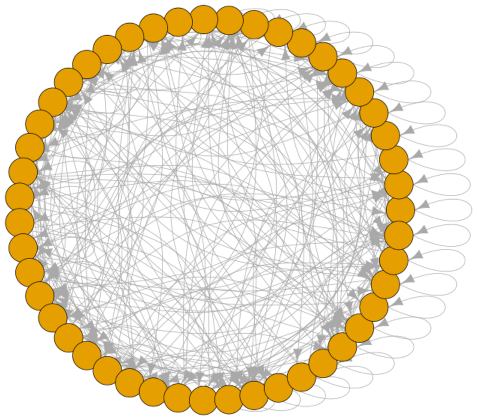

Figure 2: Graph obtained from the Gmail model.

The flattening process, which takes a fraction of a second on a low-cost PC, produces a directed multigraph with 47 vertices and 634 edges; when simplified by collapsing multiple edges between any pair of vertices, the graph includes 312 edges. Each vertex represents one of the possible combinations of simple states in any of the regions that can be active at the same time.

Similar models and corresponding graphs were produced for the other three applications.

Analysis of interaction designs

Inspection of the interaction graph is not particularly useful because even for small graphs like the one obtained from our Gmail model no particular structure is evident. Figure 2 shows a plot using a circular layout of the 47 states: because of the large number of cycles that exist among states, edges form an intricated web of possible action sequences. This in practice greatly reduces usefulness of typical network analysis metrics, such as *betweenness, eccentricity, page-rank, eigenvalue* centrality measures.

To be able to obtain results that bear upon usability, we process further the graph, first by specifying an interaction scenario using vertices and edges of the graph, and secondly by computing possible execution paths.

Specification of a scenario consists of formulating it using the language provided by the graph, i.e. deciding which is the initial state and which user actions need to be considered. For example, the following fragment of R code specifies that the scenario should include, in the given order, two occurrences of `open(conversation)`, followed by `reply`, followed by `typeBody`, etc.:

```
edgesWithLabel(gmail,"open(conversation)"),
edgesWithLabel(gmail,"open(conversation)"),
edgesWithLabel(gmail,"reply"),
edgesWithLabel(gmail,"typeBody"),
```

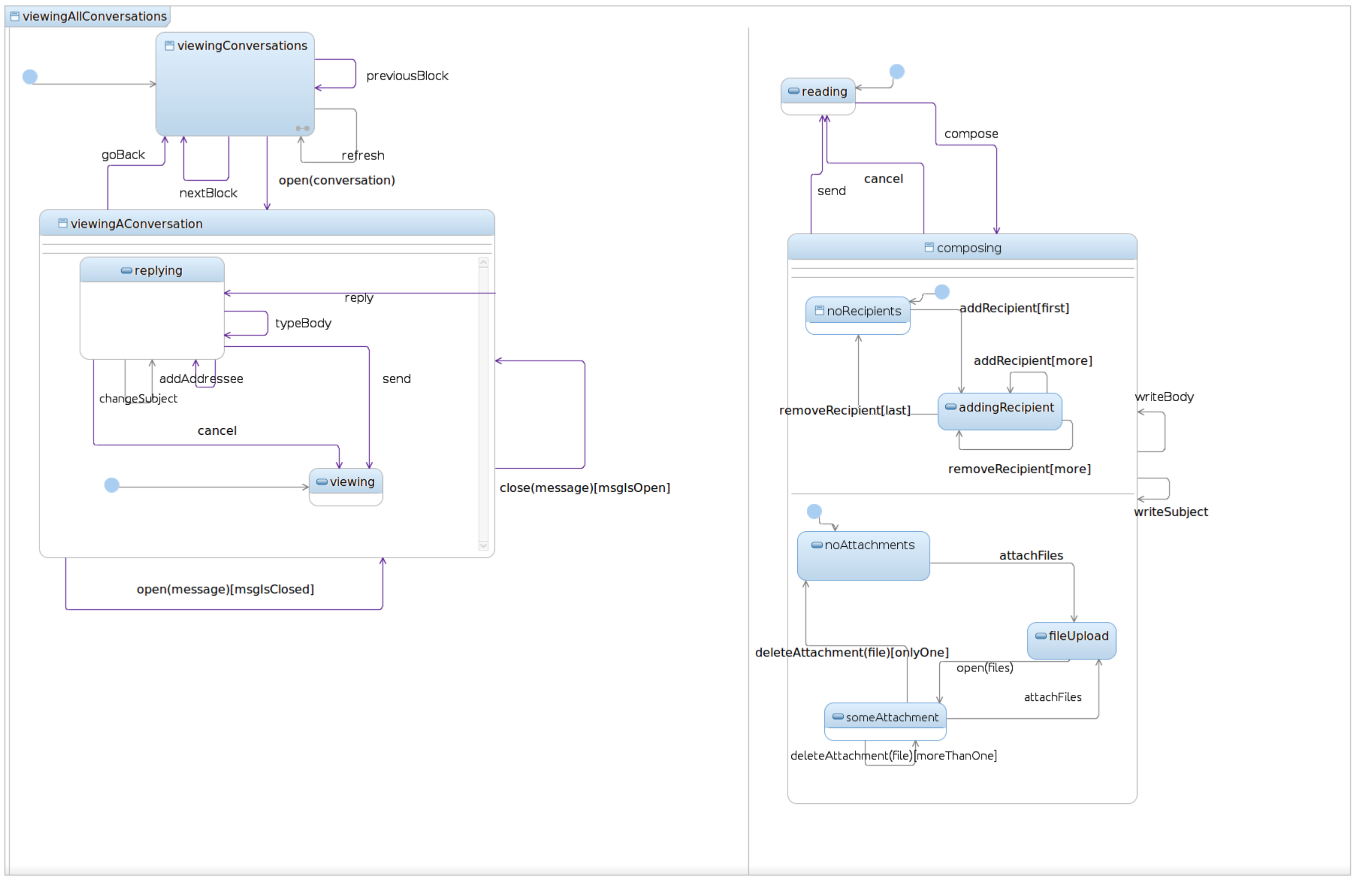

Figure 1: Part of the Gmail model.

```
edgesWithLabel(gmail,"send"),
edgesWithLabel(gmail,"compose"),
edgesWithLabel(gmail,"addRecipient"),
edgesWithLabel(gmail,"open(files)"),
edgesWithLabel(gmail,"writeBody"),
edgesWithLabel(gmail,"writeSubject"),
edgesWithLabel(gmail,"send")
```

In plain language such a scenario means opening one and then a second conversation, replying to the last message of the second one by typing the body of the response, sending it, and then composing a new message by adding a recipient, an attachment, typing the body and subject, and finally sending it.

The execution traces for such a Gmail scenario, with a detour limit of 3, consist of a graph with 474 vertices and 1753 edges. These traces entail 12 possible geodesic paths with order 0, 82 with order 1, 30 with order 2 and 33 with order 3. Figure 3 shows the number of paths obtained for the four applications, split by detour order.

Gmail has the highest number of order 0 and 1 paths, and the highest difference between the number of order 0 and 1, and between order 1 and 2 (at least 3 times as many order 0 paths than any of the other applications, and at least twice as many order 1 paths as the any of the other ones). Gmail offers 3 times as many error-free alternative paths to accomplish the scenario, which indicates that users might more easily follow one of those paths.

result

However, Gmail offers also almost twice as many order 1 paths, which means that users could be more easily induced into an erroneous path than when using another system.

result

Because the number of order 2 or 3 paths decreases, Gmail reduces therefore the "error proneness" of this UI, for executions that involve 2 or 3 errors. Notice that Roundcube has the smallest number of order 0 paths (2 of them), which means that users are not given much flexibility and freedom in carrying out correctly the scenario.

result

For none of the systems a detour leads to dead-ends preventing the user to complete the scenario.

Figure 4 shows the length of paths in the best case, i.e., when users would always choose the shortest route. Gmail offers the shortest paths across the four detour orders (for order 0 the length is 13 steps, saving more than 13% steps compared

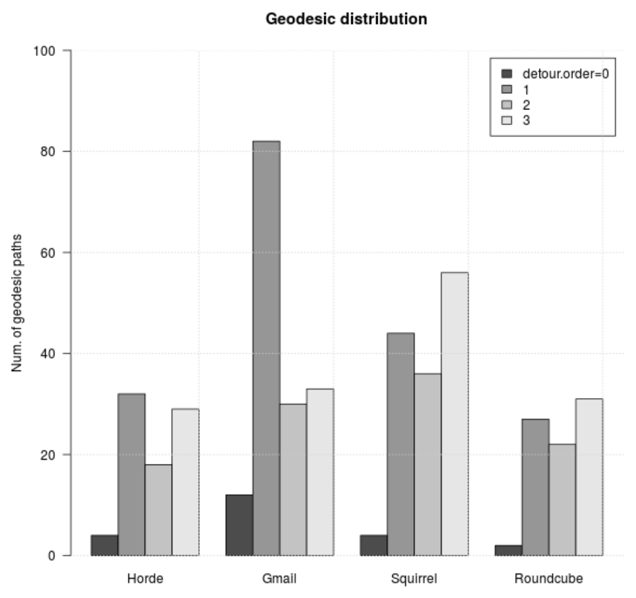

Figure 3: Number of geodesic paths split by detour order.

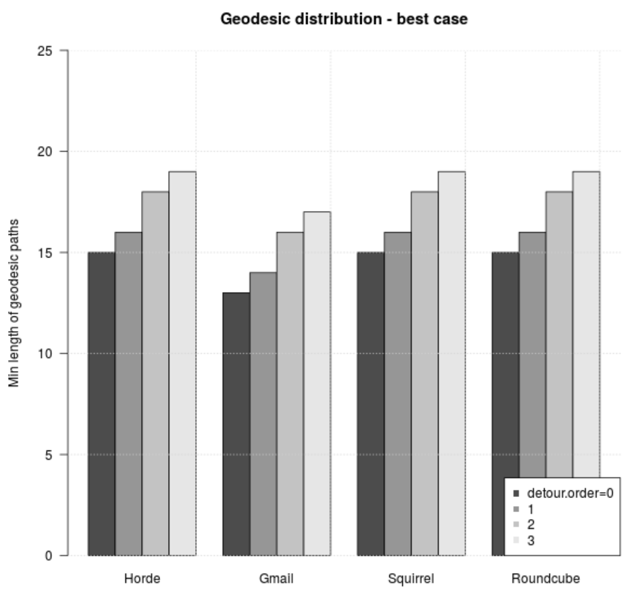

Figure 4: Minimum length of geodesic paths split by detour order.

to other applications; for order 3 the length is 17 steps; the other applications are remarkably similar among them). A plausible interpretation is therefore that Gmail not only offers many more error-free paths, but also gives the shortest ones. Users are given more flexibility and more efficiency.

result

Because also paths with order 1 or more are the shortest ones among the four applications, Gmail also makes users more efficient in recovering from errors.

result

To combine these two results, we can easily compute the frequency of paths with different length. Figure 5 shows, for each application, the frequency of order 0 paths, the frequency of paths with length less or equal to 15 (the minimum length across the four applications), and the frequency of optimal paths (the shortest ones). These values show that Gmail users have the lowest probability to hit an order 0 path (because of the relative large number of order 1 paths made available by Gmail), have the highest probability to hit a path with length 15 or less, have the lowest probability of hitting the shortest paths (10 times smaller than the best of the other applications). Thus, the flexibility and efficiency that can be exploited with Gmail are counterbalanced by the required knowledge and capability of chosing an optimal path. In particular, Gmail offers many detours of order 1 which increase flexibility for some users and might decrease effectiveness for less knowledgeable ones.

result

Another probabilistic analysis can be performed using page rank, which can be used to compute the probability that a random walk in a graph visits a certain vertex [24]. We computed the page rank (with a damping value of 5% - meaning that the random walker with probability 5% jumps to an arbitrary state and probability 95% chooses one of the actions available in the current state) for each vertex in the graph, and then added the page rank value for vertices with different detour order. Figure 6 shows the resulting probabilities.

With Gmail the probability of visiting an order 0 state is close to 70%, the lowest among the four applications. But when it comes to visiting an order 0 or 1 state, the probability increases to 87%, which is the highest. Thus, to compare Gmail with SquirrelMail, a completely random usage of SquirrelMail has 10% more probability of hitting an error-free state than Gmail. That advantage is reduced when considering order 0 and 1 paths, because with SquirrelMail the probability is 85% and Gmail it is 87%. This means that somebody with no knowledge on how to use an email front end, when using Gmail would have 10% fewer chances of carrying out the scenario without making any error, as opposed to when using SquirrelMail: SquirrelMail provides more guidance. Across the four UIs, the probability of making at most 1 error is approximately the same[3].

result

Manual inspection of the shortest paths indicates that one reason of the higher potential efficiency offered by Gmail is due to the fact that users can start composing a new message while

[3]These results are not very sensitive to the value chosen for the damping factor: differences across the four applications are very similar when d ranges in $[0.05, 0.20]$.

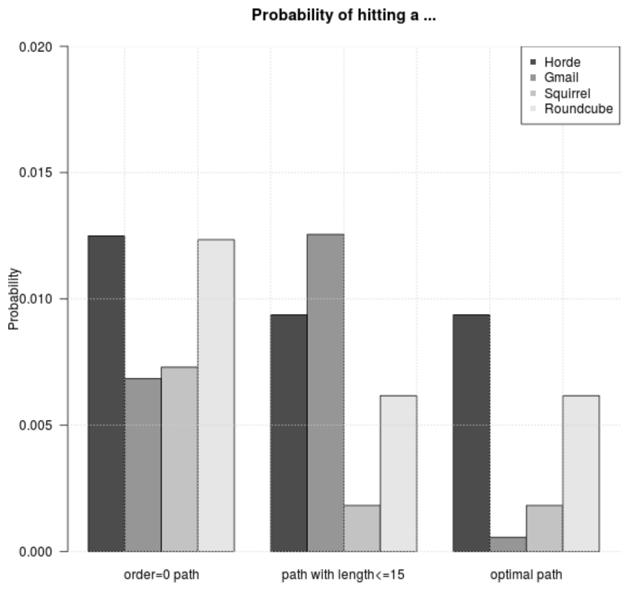

Figure 5: Frequency of an optimal path.

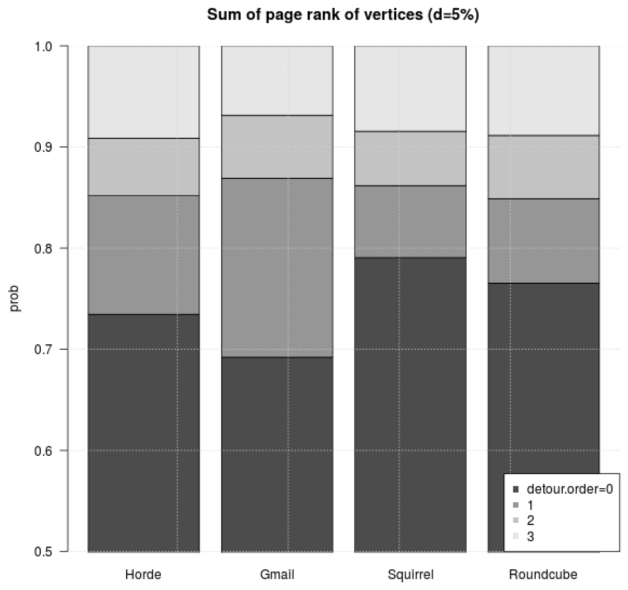

Figure 6: Probability that a random walk visits detour 0, 1, 2 or 3 states.

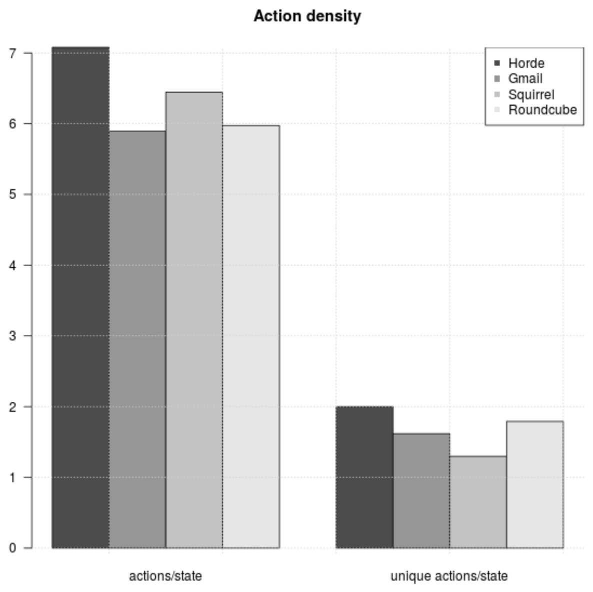

Figure 7: Action density.

reading a conversation, whereas in other applications an explicit "closing" of the reading activity has to be performed. Another reason is in the more streamlined process to attach a file: in Gmail one needs to select the file(s) and they are automatically uploaded, whereas in other applications one has to explicitly perform the uploading step after selecting them.

result

Figure 7 shows the *action density* of the four applications, in terms of average number of actions per state involved in traces, and average number of *unique* actions per state. The former is an overall measure of the number of options that are made available by a UI, the latter can be used to analyse how many *new* options the user is presented with in any state. For our examples, the values are all in the range between 5.9 actions/state in the case of Gmail and 7.1 for Horde, and 1.3 unique actions for SquirrelMail and 2 for Horde. This suggests that Gmail features a more compact design (fewer actions to do the same things), and SquirrelMail is even more compact when it comes to the different types of actions; thus it could be easier to learn.

result

Comparing scenarios

The same kind of analysis can be carried out to assess how suitable a design is for a set of different scenarios. For example, a designer might be interested in understanding what are some quantitative differences in replying to a message as opposed to composing a new one.

For Horde, it turns out that composing a new message entails many more order 0 paths (154 vs 86), and a comparable number of higher order paths. The path length in the best case is the same across the two scenarios, but in the average case composing has a length of 7.75 steps compared to 8.5 for reply. Probability of hitting an optimal path is 4 times higher for compose.

This means that users will have a twice as much large choice of correct paths when composing a new message rather than "simply" replying to a read one. On average, when composing, users could be 9% more efficient, and they are 4 times more likely to do the right things. Thus, Horde is more suitable for composing new messages than it is for replying to existing ones.

result

Adding features

Execution traces can be used also to assess what is the effect of adding a widget or feature to an existing UI. For example, we studied the cruise control features of cars. One of the examples is system S, where the driver can engage the system, and once it is engaged, speed can be increased or decreased with small or large steps. Of course the system can also be disengaged (by pressing the brake pedal, for example). System A is more elaborate, as it includes also a memory function: when it is disengaged it remembers the current speed, which can be recalled later on. There are two ways to re-engage it: one by setting a new speed, and one by recalling the previous one. In addition, if the car drives for more than 5 minutes at a higher speed than the set one, system A automatically disengages.

Thus, one possible design question is "What are the effects of adding these functionalities?" in a typical driving scenario where a speed is selected, then the system is disengaged, and then the same speed needs to be set.

System S (where we assumed that re-setting the speed is done manually by the driver with 4 actions on "up" and "down" to approximately get the same speed) leads to 1 order 0 path requiring 7 steps; there are 8 order 1 paths with average length 9.4. The probability of hitting the optimal path is 0.11; action density is 1.75, and unique action density is 1.25.

On the other hand, system A has 2 order 0 paths (average length 4.5), and 7 order 1 paths (average length 6.8). The optimal path probability is 0.06; action density is 2.2, and unique action density is 1.2.

In both cases the reasons for detours deal with the possibility of disengaging the system at the wrong moment.

Therefore we can conclude that: 1. system A makes users more efficient; 2. with A there are two possible ways to achieve the scenario, thus more flexibility is given; 3. with A the probability of doing the right thing is almost half of system S: it might be more difficult to do the right thing because more possibilities are offered; 4. system A features a more compact design, with fewer action types to be performed at each moment, being therefore potentially easier to learn and remember.

result

DISCUSSION

An important issue underlying MIGtool is the modeling effort that is needed upfront. Our experience, based on several case studies and some industrial examples, is that models do not need to be complete representations of the behavior of the application under study. By following an agile modeling approach [1], models can be easily developed by one person in less than one day, using any UML capable design tool. Even more complex models (in our experience up to 150 states and 350 transitions) can be developed and verified in 2-3 days by one person. Experience in using statecharts to model behavior of UIs is needed though. Useful suggestions are given in [15, 31].

UML state diagrams provide a very expressive language, well suited to specify behavior of UIs based on descrete events. Evidence of this can also be found in recent OMG standards, such as IFML [21]. Even though there are fundamental limits (inability to handle undo/redo's - because that goes beyond a finite state representation; inability to handle customizable toolbars - because that requires models that change at runtime; inability to handle perceptive UIs - because they are not well suited to be described in terms of descrete states), in many practical cases they can be be isolated and/or left aside).

Expressivity of the modeling language means that different designers are likely to produce different models for the same UI. As a consequence, metrics computed by MIGtool do depend on different modeling choices. It is worth mentioning, though, that because of the flattening process, several differences are reduced (for example, those dealing with using a different hierarchy of states, or with distributing differently concurrent regions across states), and such a sensitivity is correspondingly reduced. At the moment, however, we do not have hard evidence to back this claim.

Differently from other model-based approaches, such as those based on IFML, MIGtool uses *only* a model of the behavior of the UI. Designers do not need to cope with data modeling, nor with decisions dealing with presentation. In a sense, MIGtool uses only the *controller* part of the Model-View-Controller paradigm that is adopted when developing UIs. Therefore, the designer using MIGtool to perform analysis is free from other concerns that in the end affect usability, and the conclusions that are derived with MIGtool can be combined with other results *after* the analysis is performed. This also means that no effort needs to be directed on developing data and presentation specifications/implementations.

In terms of validity of conclusions obtained through MIGtool, we can say that because they are devoid of user behavior assumptions (such as preferences, skills, interpretations, ergonomic constraints) they are very general. On the other hand, predictions based on MIGtool metrics are also generic because they do not consider data and presentation aspects, nor user-related factors. For example, it is unfeasible to use MIGtool to predict the time needed by a user to complete a scenario. However, MIGtool can be used to analyze the whole interaction design and gather data to inform more specific analyses that could be performed, for example, with CogTool [16].

MIGtool is part of a suite of model-based tools for designing, analysing and testing user interfaces developed by the company ANONYMIZED. MIGtool is implemented partly in Java (model processing) and partly in R/iGraph. Development of a web-based UI of MIGtool is underway; it will be freely available for research purposes.

CONCLUSION

REFERENCES

1. S. Ambler. 2002. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley.
2. P. Ammann and J. Offutt. 2008. *Introduction to software testing*. Cambridge University Press.
3. C. Appert and M. Beaudouin-Lafon. 2008. SwingStates: adding state machines to Java and the Swing toolkit. *Software: Practice and Experience* 38, 11 (2008), 1149–1182.
4. E. Barboni, S. Conversy, D. Navarre, and P. Palanque. 2007. Model-based engineering of widgets, user applications and servers compliant with ARINC 661 specification. In *Interactive Systems. Design, Specification, and Verification*. Springer, 25–38.
5. R. Bellamy, B. John, and S. Kogan. 2011. Deploying CogTool: integrating quantitative usability assessment into real-world software development. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 691–700.
6. L. C Briand, Y. Labiche, and J. Cui. 2005. Automated support for deriving test requirements from UML statecharts. *Software & Systems Modeling* 4, 4 (2005), 399–423.
7. B. Buxton. 2007. *User Experience: Getting the Design Right and the Right Design*. Morgan Kaufmann.
8. S. Ceri, P. Fraternali, and A. Bongio. 2000. Web Modeling Language (WebML): a modeling language for designing web sites. *Computer Networks* 33 (2000), 137–157.
9. L.L. Constantine and L.A.D. Lockwood. 1999. *Software for use: a practical guide to the models and methods of usage-centered design*. Addison-Wesley.
10. J. Gómez, C. Cachero, and O. Pastor. 2001. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia* 8, 2 (2001), 26–39. DOI: http://dx.doi.org/10.1109/93.917969
11. D. Harel. 2009. Statecharts in the making: a personal account. *CACM* 52, 3 (March 2009), 67–75.
12. M. Heymann and A. Degani. 2007. Formal analysis and automatic generation of user interfaces: approach, methodology, and an algorithm. *Human Factors: The Journal of the Human Factors and Ergonomics Society* 49, 2 (2007), 311–330.
13. D. M Hilbert and D.F. Redmiles. 2000. Extracting usability information from user interface events. *ACM Computing Surveys (CSUR)* 32, 4 (2000), 384–421.
14. K. Hornbæk. 2006. Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human-Computer Studies* 64, 2 (2006), 79–102.
15. Ian Horrocks. 1999. *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
16. B.E. John, K. Prevas, D.D. Salvucci, and K. Koedinger. 2004. Predictive human performance modeling made easy. In *Proc. of the Int. Conf. on Human Factors in Computing Systems*. ACM, New York, NY, USA, 455–462.
17. N. Koch and A. Kraus. 2002. The expressive power of UML-based web engineering. In *Second International Workshop on Web-Oriented Software Technology (IWWOST02)*, D. Schwabe, O. Pastor, G. Rossi, and L. Olsina (Eds.). 105–199.
18. M. McCurdy, C. Connors, G. Pyrzak, B. Kanefsky, and A. Vera. 2006. Breaking the fidelity barrier: an examination of our current characterization of prototypes and an example of a mixed-fidelity success. In *CHI 2006*. ACM, ACM Press, New York, NY, 1233–1242.
19. S. Meliá, J. Gómez, S. Pérez, and O. Díaz. 2008. A Model-Driven Development for GWT- Based Rich Internet Applications with OOH4RIA. In *Proc. 8th Int'l Conf. Web Eng. (ICWE 2008)*. IEEE CS Press, 13–23.
20. G. Mori, F. Paternò, and C. Santoro. 2002. CTTE: Support for Developing and Analysing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering* 28, 8 (August 2002), 797–813.
21. OMG. 2013. *Interaction Flow Modeling Language (IFML), FTF – Beta 1* (omg document number: ptc/2013-03-08 ed.). Technical Report. OMG. http://www.omg.org/spec/IFML/1.0
22. Object Management Group OMG. 2015. OMG Unified Modeling Language (OMG UML) Version 2.5. http://www.omg.org/spec/UML/2.5. (March 2015). http://www.omg.org/spec/UML/2.5/PDF
23. M. Otter and H. Johnson. 2000. Lost in hyperspace: metrics and mental models. *Interacting with Computers* 13, 1 (2000), 1–40.
24. L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. The PageRank citation ranking: bringing order to the Web. (1999).
25. D.L. Parnas. 1969. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *ACM '69 Proc. of the 1969 24th National Conference*. ACM.
26. P. Pinhero da Silva and N.W. Paton. 2003. User Interface Modeling in UMLi. *IEEE Software* (2003), 62–69.

27. J. Preece, Y. Rogers, and H. Sharp. 2002. *Interaction design*. John Wiley and Sons.

28. J. Rubin and D. Chisnell. 2008. *Handbook of Usability Testing* (second ed.). Wiley.

29. M. Samek. 2009. *Practical UML Statecharts in C/C++*. Elsevier.

30. P.A. Smith. 1996. Towards a practical measure of hypertext usability. *Interacting with Computers* 8, 4 (1996), 365–381.

31. H. Thimbleby. 2007. *Press on: principles of interaction programming*. The MIT Press.

32. H. Thimbleby, P. Cairns, and M. Jones. 2001. Usability analysis with Markov models. *ACM Transactions on Computer-Human Interaction (TOCHI)* 8, 2 (2001), 99–132.

33. H. Thimbleby and P. Oladimeji. 2009. Social Network Analysis and Interactive Device Design Analysis. In *Proc. of Engineering Interactive Computing Systems 2009*. ACM Press, 91–100.