PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 47, NO. 3–4, PP. 297–309 (2003)

INVARIANT USER INTERFACES

Szabolcs BARANYI, Károly HERCEGFI and Károly TILLY

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Magyar Tudósok körútja 1, Budapest, Hungary
e-mail: baranyi@mit.bme.hu

Received: Oct. 1, 2003

Abstract

In this article the term *invariant user interface* is introduced for a generic, stable backbone of all user interfaces, which contain a set of predefined elements and rules to build complex interactive systems. Invariant user interfaces specify *fix points in using information systems.* We argue that in interfaces of complex software applications such fix points are increasingly necessary. Based upon psychological assumptions and results of human computer interaction (HCI) studies, the necessity and benefits of invariance is shown, among others increased efficiency, enhanced reliability of use and decreased cost of software ownership. In this article invariance properties of state-of-the-art user interfaces are summarized, and a simple interaction model is introduced. Types and limits of invariance is defined using this model, and a set of fundamental criteria is characterized that invariant interfaces must meet.

Keywords: user interfaces, invariance criteria, mental models, abstract interfaces, abstract devices.

1. Introduction

Humans need fix points which control their behavior and help them in planning actions for unforeseen situations. These fix points are rules and assumptions which allow to interact with our environments, and to predict and react to events. In the real world such fundamental rules are established upon social, biological or physical laws (ATKINSON [2]). A *user interface* is considered *invariant* if its operation is built around a set of predefined *fix points of using information systems.*

Users can nowadays access computer system services through platform, device and application-specific interfaces. An average computer user must be able to work daily in different environments like desktop computers, laptops, palmtops or handies. In the case of direct manipulation interfaces, metaphors of windows and widgets span different operating system environments and are well known and accepted by large user communities. On the other hand, elements having higher complexity or higher level semantics like commands, navigation sequences or important user interface component layouts are not invariant even inside a single operating system.

Designing user interfaces is based on tradition and guidelines of large software vendors and consortiums like APPLE [1], IBM, the OSF group [8] or or Microsoft [13]. These guidelines define metaphors and the widely used interface elements

whith semantics widely known and accepted by the majority of users. Recommendations comprise only widget-level visual design and some rules for widely used menu items and shortcuts.

Certain criteria of consistency are contained in these recommendations, but these are only based on tradition. It is obvious that currently no theory of invariance exists, and the criteria and applicability limits of invariance are also unknown. The lack of invariance leads to a set of significant problems.

The first problem is that the use of applications must be learned and relearned for each domain, environment, vendor and even software version. Large software owner firms spend millions of dollars year by year for software training of employees.

The second problem is that relearning decreases efficiency. A problem of relearning and the importance of standards for commands are reported in KEITH [11]. A number of independent software ergonomic research studies show [7] that a user can handle a computer system *efficiently* if he can use a software product without significant mental load. It is also shown that mental load is decreased through practice and experience IZSÓ [9].

The third problem is that relearning increases the probability of user errors. Thus, the stability of applicable knowledge at the user interface level is required to increase reliability, which is especially crucial in cases like safety critical systems where human errors have severe consequences.

It can be concluded that potential benefits of invariance in user interfaces include increased efficiency and reliability of use, a decreased burden of learning with no relearning in the optimal case and thus a decreased cost of software ownership.

Furthermore, invariant user interfaces offer the possibility of creating a generic symbol system and concepts that future communities of users can learn at grammar school and apply them for a lifetime, without relearning or further customizing their knowledge. So our vision could be summed up in a simple sentence: *Learn once, use for a lifetime.*

2. A Model of Invariant Interaction

For the purposes of introducing and discussing the problems and elements of invariance, a user oriented model of information systems is applied. See *Fig.1*. This model is based on the Arch reference model of interactive systems [3]. According to this model a computer system offers a set of *services* to the user. The user can formulate *requests* for the execution of services above a selected set of *contents*. User requests are executed by appropriate providers (servers) and the results are passed back to the user.

The *user space* is representing concepts as the user sees them, namely interface elements, computer system services, contents and devices. The *implementation space* contains software and hardware modules, which can execute services, implement user interface components and realize hardware IO devices. In the following

sections we provide evidence that the organization of concepts and the elements of the user space are crucial in defining fix points of information systems.

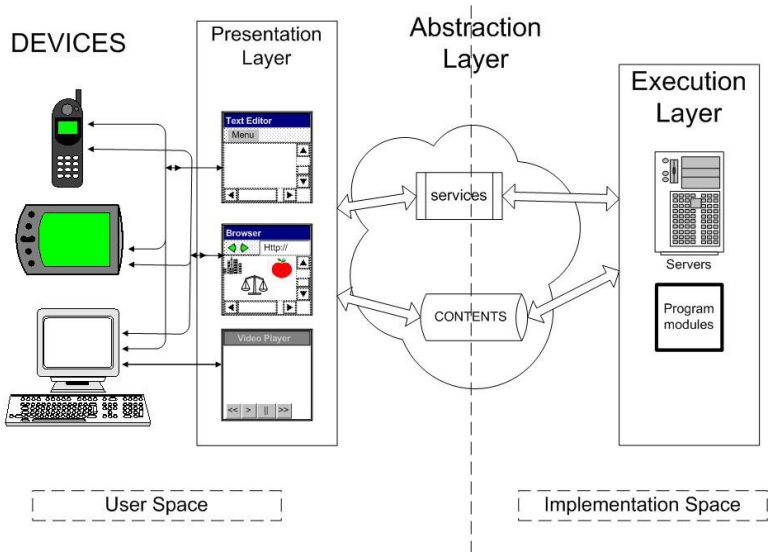

Fig. 1. User-oriented model of information systems

This model assumes that the *user* solves his problems with the help of a computer system, which executes a set of *services* upon user requests. Service requests are executed by *providers*, which assign a service to a list of arguments in the form of pieces of *contents*. Users can request *services* through *user interface components*, which also have the responsibility to display the results produced by the computer system. User interface components are accessed through *devices*.

The *presentation layer* represents concepts as the user sees them, namely interface components, services and contents. Interface components in the presentation layer are mapped to *abstract devices* which are introduced to describe a set of higher level functionalities of pieces of real hardware. Invariant usage of functions on various hardware products is called device independence, which is subject to extensive research and standardization W3C device independence group [19].

The *execution layer* contains pieces of software, which can execute services. Components in the execution layer must meet abstract service specifications, and can be flexibly combined to form compound service structures.

The most important precondition of invariant interaction is a set of information, independent of both the way of presentation and execution of services. This is encoded into the *abstraction layer*. It separates the semantics and implementation of services and contents from their presentation and execution, and defines a mapping between elements of the presentation layer and the execution layer.

The elements of the abstraction layer are *abstract services* and *content types*. When implementing invariant interfaces, the abstraction layer serves as the basis of

defining protocols between presentation and execution components.

Services are offered through interface components in the form of *commands*. Interface components also display *pieces of* contents, which are either arguments of service requests or results of service execution. When formulating requests, the user selects arguments, selects a command, and activates it. These steps require *navigation* (to find the place of a command or piece of content in the interface), *recognition* (to realize that a command represents a desired service or that a piece of content represents a desired argument) and *activation* (to formulate a request and to start its execution).

3. The Importance of Abstraction and Separation

A major problem of current software applications is the large number of application environments that a user must learn. In the following # denotes set arity, while S is the set of abstract services a user requires, C is the set of content types the user can interpret, E is the set of interface elements the user can apply to formulate requests and A is the set of application dependent actions, a user must learn to formulate necessary requests. So a user can access $\#S$ different services above $\#C$ different types of contents through $\#E$ different user interface elements using $\#A$ different actions that a user must learn to use different interfaces.

According to *Fig. 1* $\#A >= \#S + \#C$ (1), since the user must at least learn the meanings of services and content types to be able to formulate requests. In the case of an ideal invariant interaction $\#A = \#S + \#C + \#E$ (2), any service appears and can be accessed the same way, independently of the content and the user interface elements. The most pessimistic extreme is $\#A = \#S\#C\#E$ (3), where any service with a specific semantics appears differently for each kind of contents, can be accessed through different user interface elements in each application environments.

State of the art is far from (2), but surprisingly is close to (3). It means that current user interfaces are ad-hoc, application dependent ones, and the presentation of services (mostly appearing as commands) strongly depend on the contents they operate on. The major precondition of approaching (2) is that services and contents can be combined orthogonally, so the abstraction layer must define the semantics of abstract services independently of the semantics of content types (data objects). Also user interface components must be orthogonally associated to services and must be able to present any appropriate content types. Notice that a fully independent orthogonal combination of services, content types and interface components is a theoretical optimum.

4. Invariance in the User Space

The following sections specify important criteria for arranging the navigation/recognition/activation process. The goal is to identify major blocks and structures

that serve as a guideline to determine fix points of user interfaces based upon generic regularities of users' thinking.

The user space is analyzed according to Rasmussen's model [15], which is widely used for modelling the cognitive levels of human information processing. It assumes that human activities are driven by goals. To achieve goals, humans sense external signals and react to them by answers in the scope of the desired goals. Actions are organized into a hierarchical structure, where higher levels are composed of lower level entities and, at the end, elementary actions.

The lowest level includes *skill based* actions, which are sequences of *sensomotor* activities without conscious control. Input events are recognized and some elementary actions executed automatically in the scope of goals. Higher level processes are built on skill-based reactions. Motor level activities are kinds of *activations* like starting command execution through shortcuts, menus or keyboard commands. Typing/selecting texts in an editor box, most frequent kinds of *navigations* among interface components or pieces of contents are also of motor level.

The next level describes *rule-based behavior.* It incorporates complex or not so often practiced actions, which need human attention to execute. This level can be characterized by well-known rules associated to signals. When a signal is recognized, a rule is selected according to the goals, and sequences of elementary actions are executed as a consequence of the rule. The major difference between rule-based and motor level actions is conscious control. Signs mean much more than signals, they require attention to recognize raw input and to associate it with a concept, like seeing an icon and identifying a command represented by the icon.

The *knowledge-based* level of behavior is the highest. At this level we need to analyze a situation described by signs, there are no well-known schemes of reaction, so a plan must be created. Notice that this is the only level which is inherently variant, so no invariance rules are applicable here.

Beyond this model we rely on the general model of Human Information Processing, and rules of learning. As well-known, the cognitive memory is divided into short-term (STM) and long-term memory (LTM). The storage capacity, information organization of the LTM, the learning and retrieval processes, and features like shadowing are important from the point of invariance [2].

5. Analyzing the Sensomotor Level

The generalized power law equation [5] points out that the probability of retrieving data with given attributes depends on the number of repeated usage of the appropriate object. From the point of view of invariance it means that a common function used more times causes a more certain usage.

It takes a long time and practice to learn skill-based actions, and it is hard to relearn new ones. Errors caused by an old action instead of a new one are highly significant in safety critical systems.

Errors at the motor level are slips or lapses [16]. They can occur when

something disturbs or confuses motor level actions. The probability of this type of error increases when a new action for the same goal has to be learnt.

To analyze users' performance, the family of GOMS (Goals Operators Methods Selection rules) models was suggested to provide an engineering aspect of human performance. These models divide processing into three major subsystems, the *perceptual system* the *cognitive system and the motor system* which act parallel. Detailed description of GOMS and related models are described in [5, 4]. It is useful for measuring motor level action and performance and there are different variants on the same concepts for different purposes.

Our first conclusion is that *any elements of the user space related to the skill-based level must be invariant and common for all applications.* Invariance is mandatory for lower level building blocks of the interface hierarchy, and should be valid up to the highest possible level.

6. Analyzing the Rule-Based Level

Information organization plays an important role in learning. Independent studies show [13, 17] that users learn much faster and more efficiently if they have a model to organize the information. It shows that we need to organize our invariant entities into ontologies, and we have to define a model for these ontologies. Experts obviously have a well-organized hierarchical mental model about the system, and they use top-down strategies in problem solving [12].

Information retrieval is context-sensitive and pieces of information learned together facilitate retrieval. User interface design patterns, layouts act as context of work and have great importance in rule-based actions. If a user can identify well-known interface elements on a layout, he will suppose certain rules to work. For example, if a commander window appears with two panels, and lists of files on each panel, experienced users will certainly have a Windows Commander style interface in mind, so they may assume that the Tab key switches between panels, and the F5 key activates a copy operation of the selected files. When an interface pattern appears, there are rules associated to it in the user's mind, like relationships between viewers and services. Abstract viewers can also be categorized based upon a set of predefined interface patterns which are more complex than widgets, and they can be embedded into each other, forming compound interface structures Jenifer TIDWELL [10]. We think that the most informative level of user interfaces are not widgets but interface layout patterns.

As a result of relearning, it is possible that a wrong but frequently used rule will be activated instead of the wanted one because of interference in information retrieval [2].

Based on GEMS [16], errors at the rule-based level include faulty signal recognition, the application of a wrong rule or the application of an out-of-date but strong rule. Invariance cannot deal with the first two causes but it can decrease the number of twisted rule applications caused by relearning. It has an importance in

stress situations and in safety critical systems.

Research results mentioned above support the idea that modelling and organizing pieces of information help users to use and learn them better. In our model, users are professionals knowing their domain and having exact knowledge about the services they require. To aid them in finding and selecting the right service, it is reasonable to organize them into a Service Ontology.

Our second conclusion is that *services must be organized into service ontologies in a reproducible and extensible way. There are complex interface structures, which can also be recognized by users, and thus, they can be made invariant and organized into viewer ontologies.*

7. Building Ontologies

When building ontologies the basic task is to classify services, viewers and content types present in current computer systems. The basis of classification is the meaning of services, viewers and pieces of contents for the user. Implementation details are unimportant, the only significant issue is the set of requirements (preconditions) for the user of applying a certain item and the set of results. Classification is based upon a formal description of semantics for the given entities. We found that an ETAG (Extended Task Action Grammar) – based formal description was suitable to describe the user's model of services (G. HAAN 2000 [22]). In the scope of invariance the canonical basis introduced in ETAG has a great importance, since it must be invariant and extensible. In the following section a canonical basis is constructed and examples are provided for describing concrete services. Notice that ETAG addresses user interface analysis and design which is a much broader domain than service classification, so just a limited subset of ETAG is used for semantic service specifications.

In a complex computer system, users perform operations above a C set of contents. The computer system has the ability to execute these operations, and to generate a set of contents as a result. A service can be defined as an s function with n arguments of the form $s : \times^n C \rightarrow C$, where $\times^n$ denotes an n-ary Cartesian product.

Let us introduce the notion of *Service Semantics Specifications* (SSS, denoted by S^3). An S^3 is defined by the following five tuples: $S^3 = \{s, P, A, C, E\}$, where s is a service; P is a finite alphabet whose symbols can be used as *predicates* in first-order logic statements; A is a finite alphabet whose symbols can be used as *attributes*, which serve as variables of first-order logic predicates; C is the *precondition* of s defined by a set of first-order logic statements over P and A, i.e. C can contain predicates from C with arguments taken from A; E is the *effect* of s. Like C, E is defined by a set of first-order logic statements over P and A, i.e. E can contain predicates from E with arguments taken from A.

The set of predicates are formulated based upon an object-oriented world model of service execution where operations are performed above sets of *objects*

having their own *types* (*classes*), *identities*, *values*, *lifetime* and *properties*. Any service is an operation which can be performed above a set of objects with given types, and produces certain objects of given types.

According to the above object-oriented service model, a set of predicates were defined. The following is a small subset of the initial predicates:

- *argument(o)*. Object *o* is an *argument* of a service.
- *create(o)*. Object *o* is newly created as an instance of a class.
- *current(o,env)*. Object *o* is an active (tipically selected) object of object *env* specifying an environment.
- *reference(o,r)*. Object *o* can be accessed through reference *r*. Even if in a concrete predicate *o* and *r* are the same, *o* is and object with an identity and value, while *r* is just a reference to *o*. It also means that any object can serve as a reference.
- *storage(o), storage(o,st)*. Object *o* is stored at location *st*. The value of *st* is one of {TRANSIENT, PERSISTENT, REMOTE, LOCAL, PRINT}. If *st* is omitted, the storage of *o* is TRANSIENT.
- *type(o,t)*. The type of object *o* is *t*. The value of *t* is one of {EXPRESSION, OBJECT}.
- *type(t)*. Declares a *t* type variable whose values can take the name of any type.
- *value(o,v)*. The value of object *o* is v.

There are some additional predicates related to viewers, which are device-independent abstract (high-level) objects for presenting pieces of contents to the user, and for handling user interaction using various devices and events. Viewers can be associated to services using type constraints. So we do not define a concrete viewer or viewer type for a service, but we may specify rules that describe a functionality that a presentation object must meet.

We analyzed a search program called 'Copernic Agent Professional' [6] using the above mentioned methods and we built an ontology of its services. As an example, *Fig. 2* presents the S^3 description of the Copernic Agent *Save page as* service

8. Generating Service Ontologies

First the S^3 specifications of the services must be described. In the next step the S^3 specifications are converted to so-called *service graphs*, whose nodes are labeled by either predicates or attributes of the S^3, and there is an edge between a *p* predicate node and an *a* attribute node whenever predicate *p* refers to attribute *a*.

As an implementation concept graph (CG) based representations were applied (SOWA [21]), which can express meaning in a form that is logically precise and computationally tractable, furthermore are easy to generate from predicate-based descriptions.

```
service('save page as', 'Copernic')
    Precondition:
        Argument(*wd);
        Storage(?wd,''REMOTE'');
        Current(*rwd,results);
        Reference(wd,?rwd);
        Argument(*name);
        Type(*t);
    Effect:
        Create(?o);
        Type(?o,?t);
        Storage(?o,''LOCAL'');
        Reference(?name,?o);
        Value(?o,?wd);
```

Fig. 2. A sample S3 description: The Copernic Agent *Save page as* service

The equivalent Concept graph description of the Copernic Agent *Save page as* service is shown *Fig. 3*. Concept [*TR*: *type-name*] refers to a sub-concept, which expresses a type constraint referenced by *type-name*.

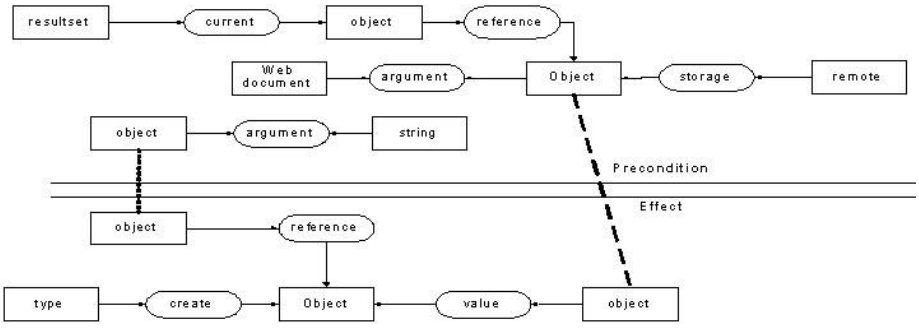

Fig. 3. The graph representation of the *Save page as* service of Copernic.

The generation of ontologies over elements of sets always require an ordering relation. This ordering relation is based upon *isomorph intersections* of pairs of service graphs. An isomorph intersection is a maximal isomorph subgraph presents in both service graphs. An isomorph intersection shows that two service specifications contain equivalent parts. There are tools, which help building and analyzing concept graphs. During the analysis a hierarchical ontology of services can be generated using graph-matching algorithms. Another advantage of concept graph like representations is that there are methods and automatic tools for generating the minimal lattice of service ontologies based upon formal concept analysis (FCA) developed by Bernhard GANTER and Rudolf WILLE [20].

9. Implementation

As discussed above, a major concern is orthogonal combination of services, contents and user interface elements. To turn this theory into practice, an architecture is needed which supports dynamic loading of user interface elements and services providers and has the functionality to combine them together according to the constraints defined in abstract service descriptions (see *Fig.4*).

It is important to notice that component-based software environments like .NET or J2EE provide a sound technical background for implementing such systems.

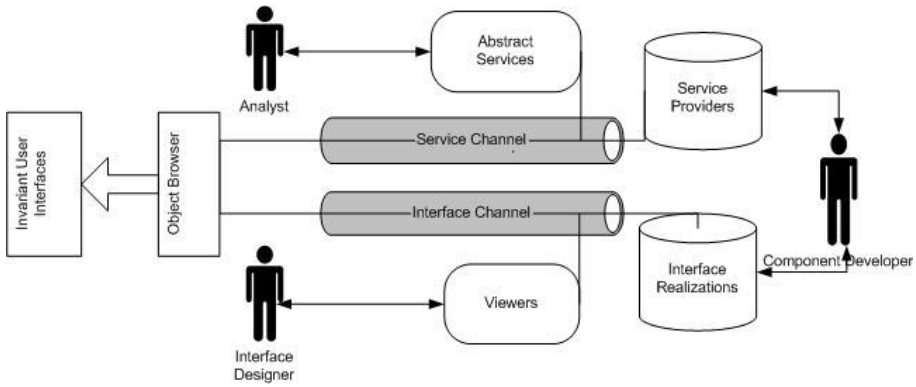

Fig. 4. The architecture of the GENUIN invariance user interface system

When a user starts to work with an interface, he refers to the name of an interface definition which is basically an application in the invariant user interface environment. The invariant interface is a working set of software and interface components loaded and executed according to an interface definition. The system determines the device that the user interacts with (like command line, GUI, WEB browser or handy). Depending on the device type and the required viewers, the interface realization is retreived from a repository and displayed by the actual device using a presentation framework called the object browser. The object browser acts as an assembly for user interface elements, it has the responsibility to dynamically load interfaces delivered through the interface channel. Invariant user interfaces can be used at any devices that implement the object browser, which, in general, is a main window or a WEB/WAP browser. The system processes user inputs and forwards requests to providers through the *service channel*, which has the responsibility to select the appropriate provider. Providers have no user interface, and are able to perform predefined services on behalf of the user. Providers accept requests originating from viewers, they execute requests, and return execution results to the appropriate interface element. The separation between services and user interfaces is clear. Services do not take care of implementation and specialties of user interfaces, while the user does not know anything about service implementations.

The outlined architecture is inherently distributed, i.e. interface components, providers can reside physically anywhere in a computer network. Information flow between system components takes place through the interface channel and the service channel. Since viewers must run at the client machine in the environment of the object browser, their code must be physically downloaded to the client side. This can be done through the interface channel. The execution of requests, in general, does not require downloading the appropriate providers, though messages must be passed for specifying the request, and results must be received. This is accomplished through the service channel. The idea is to build software components with well-defined interfaces. Larger programs can be composed of elementary components in the Lego style. They recommend a uniform description of component interfaces using preconditions and postconditions. An algorithm is provided for selecting interface implementations. In our case we follow the same scheme while separating definitions from implementation. Our specifications are abstract services defined based on user needs. At this point we only deal with services that directly appear at the user level, so we do not care about hidden system components. For a detailed discussion of the GENUIN architecture see [18].

10. Conclusions and Further Work

In this article the elements, benefits, and limitations of invariance in user interfaces were analyzed based upon related research results. We can conclude that invariance is a highly desirable and useful property of user interfaces, though there is a long way ahead till it will be reached. Necessary steps can be characterized by our model.

We studied user interaction at different cognitive levels and specified two major ways of invariance.

Our first important conclusion is that any elements of the user space related to the skill-based level must be invariant and common for every applications.

The second one is that there are complex interface structures which can also be recognized by users, and thus can be made invariant and organized into a Viewer Ontology.

The first step is to clearly separate interface entities with a semantic meaning to the user. Important entities are *abstract services*, *abstract contents* and *abstract viewers*. This requires a careful analysis of the specified entities in current computer systems. We did certain initial steps towards analyzing abstract services. The goal is to define a set of orthogonal items which can be combined in an arbitrary way. So the number of entities a user must learn can be drastically decreased.

It is important to separate presentation from execution, which means that invariant interfaces will not depend on different application environments or operating systems. Fortunately, this requirement can be fulfilled using state-of-the-art distributed object technologies. We implemented an experimental architecture described in [18].

The next issue is to associate semantic entities with appropriate presentation attributes. So abstract services must be associated with standard access attributes like key combinations, command strings, button and menu labels, icons or others, while abstract viewers must be associated with predefined layout schemes, with specific gestures, basic navigation sequences and use cases.

The last step is to organize the entities into fixed ontologies to assure invariance of features related to navigation like menu structures and access paths of different dialogs.

Notice that there are higher-level operations which can change between application domains. These introduce the notion of *domain-dependent invariance*. Domain-specific features are related to *domain analysis*, which is a relatively new approach, and it seems to have high significance in invariant interfaces.

A further important remark is that invariance criteria are not rigid standards, but they just define a set of necessary conditions for user interfaces which still allow different designs.

The limits of our invariance rules seem to be close to the applicability limits of the GOMS model.

The answers to the previously mentioned questions will definitely lead to standards. At the beginning it will be interesting to decide which nowadays used commands and actions will take part in it. In order to find an answer, we are analyzing existing interfaces and user interface building tools.

References

[1] Aqua Human Interface Guidelines: Apple Computer Inc. 2002.
[2] ATKINSON, R. L. et. al., *Hilgard's Introduction to Psychology*, Harcourt Brace Collage Publishers Fort Worth, 1996.
[3] BASS, L. – COUATZ, J. – UNGER, C., A Reference Model for Interactive System Construction, 1992.
[4] JOHN, B. E. – KIERAS, D. E., Using GOMS for User Interface Design and Evaluation: Which Technique? *ACM Trasactions on Computer-Human Interaction*, **3** No. 4 (1996), pp. 287–319.
[5] CARD, S. K. – MORAN, T. P. – NEWELL, A., *The Psychology of Human–Computer Interaction*, Lawrence Erlbaum Associates, 1983.
[6] Copernic Terchnologies Inc., CopernicAgent, http://www.copernic.com/en/index.html.
[7] HELANDER, M. – LANDUER, T., *The Handbook of Human–Computer Interaction.* North Holland, Amsterdam, 1996.
[8] IBM Ease of Use User Interface Architecture 2nd ed.:IBM Corp. 2001.
[9] IZSÓ, L. – ZIJLSTRA, F., Efficiency in Work: An Approach to Interface Evaluation and Design, *Proceedings of the 8th European Conference on Organizational Psychology*, (1997), p. 39.
[10] TIDWELL, J., COMMON GROUND: A Pattern Language for Human–Computer Interface Design, http://www.mit.edu/^jtidwell/common_ground.html.
[11] BUTLER, K. A., Usability Engineering Turns 10, *Interactions*, **3** No. 1 (1996), pp. 58–75.
[12] LARKIN, J. H. – MCDERMOTT, D. – SIMON, D. P – SIMON, H. A., Expert and Novice Performance in Solving Physics Problem, *Science*, **208** (1980), pp. 1335–1342.
[13] Official Guidelines for User Interface Developers and Designers: Microsoft Corporation 2002; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/welcome.asp.

[14] NILSEN, E. – JONG, H. – OLSON, J. S. – POLSON, P. G., Method Engineering: From Data to Model to Practice, *Proceedings of the CHI'92*, pp. 313–320.
[15] RASMUSSEN, J., Skills, Rules, and Knowledge; Signals Signs, and Other Distinction in Human Performance Models, *IEEE Transactions on Systems Man, and Cybernetics*, Vol. Smc-13, No. 3, (1983).
[16] REASON, J., *Human Error*, Cambridge University Press, 1994.
[17] FEIN, R. M. – OLSON, G. M. – OLSON, J. S., A Mental Model can help with Learning to Operate a Complex Device, *Conference on Human Factors and Computing Systems*, (1993), pp. 157–158.
[18] TILLY, K., Genuin: An Application Programming Environment for Invariant User Interfaces, Technical Report, Budapest University of Technology and Economics, Department of Measurement and Information Systems.
[19] W3C Working Draft, September 2001, Device Independence Principles http://www.w3.org/TR/2001/WD-di-princ-20010918/.
[20] GANTER, B. – WILLE, R., *Formal Concept Analysis · Mathematical Foundations*, Springer, ISBN: 3-540-62771-5.
[21] SOWA, J. F., *Conceptual Graphs*, draft proposed American National Standard, NCITS.T2/98-003.
[22] DE HAAN, G., ETAG, a Formal Model of Competence Knowledge for User Interface Design, PhD thesis, Department of Mathematics and Computer Science, Free University Amsterdam, 2000.