CatchIt, a Development Environment for Transparent Usability Testing

Gaëlle Calvary **Joëlle Coutaz**

CLIPS-IMAG
BP 53
F-38041 Grenoble Cedex 9, France
Phone: +33 4 76 51 48 54
{Gaelle.Calvary, Joelle.Coutaz}@imag.fr

ABSTRACT

No existing tool addresses both the development and evaluation process for interactive systems. Current evaluation tools support either the predictive or the experimental approaches to usability testing, and automated tools that make possible experimental evaluation impose a tricky manual instrumentation of the existing code. In this paper we describe CatchIt, a development and usability testing environment, to respond to these limitations using work domain descriptions as the central foundation. Within CatchIt, work domain descriptions define a reference model that serves two complementary purposes: the model can be used as reusable code from which a new application can be built or it can be used as a standard against which new applications can be tested. Evaluation, whether it be predictive or experimental, consists of the automated detection of deviations of the new application from the norm defined by the reference model. The reference model evolves over time based on the accumulation of past experience. As such, it supports traceability and evolutivity of the corresponding work domain.

Keywords

Usability testing, reusability, traceability, model-based specifications.

INTRODUCTION

In recent years, usability testing has been the subject of growing interest, resulting in an explosion of methods and techniques. Although the HCI scientific and industrial communities acknowledge the merits of user interface evaluation, usability testing is still considered to be time consuming, unreliable, and based on expert experience [18]. In practice, usability testing is, at best, summative, i.e., performed at the end of the development process under temporal and financial pressure. For critical systems such as those described in [15], the adequation of man-machine interaction is not a luxury, but a necessity [14]. The increasing complexity of on-board systems [3] exacerbated by tight competition, militate in favor of rigourous methods. Previous experience shows that domain knowledge provides sound foundations for such methods.

We observe that knowledge about the work domain is used explicitly at the specification and design stages of the development process; it motivates technical decisions; it shapes the definition of the final product. However, important information about the work domain is lost on the way between the early phases of the development process and the implementation phases. As a result, the design rationale may not be understood by or accessible to the software development team [26]. Deviations from the intended design may occur. In addition, developers tend to view interaction as a set of functional issues rather than as operational concerns situated in a context of use. These problems result primarily from the lack of continuity in the development process. They may be solved by a better capitalization of knowledge and traceability along the development process.

Capitalization can be improved through the formal description of the work domain. Although the cost/benefit of formal descriptions is still unclear to most designers [16], reusability offers an attractive way for alleviating costs. The industrial trend for specialization, the push of the competition, and the advent of object-oriented technology, all motivate the concept of reusability. Work domain descriptions capture the current expertise of a company for that particular domain. Reusability of work domain descriptions supports the accumulation of expertise across time. In turn, accumulation of expertise increases competitivity. For critical systems whose complexity continue to increase, reusability of work domain descriptions is becoming vital.

As shown in Figure 1, work domain descriptions can be exploited in many ways: as a reference framework for designing new applications within that work domain, as a mechanism for supporting traceability and reusability, as the grounding material for usability testing.

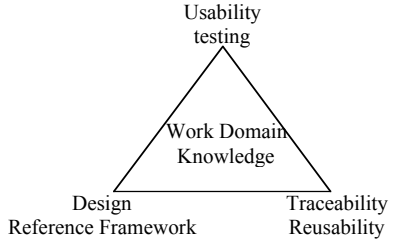

Figure 1. Work domain knowledge as a facility for usability testing, reusability, traceability, and a reference framework for new designs.

This article describes CatchIt [6], a Critic-based Automatic and Transparent tool for Computer Human Interaction Testing. As discussed in the next section, no tool so far addresses both the development and the evaluation process of interactive systems. With regard to evaluation, current tools support either the predictive or the experimental approaches to usability testing. Automated tools that support experimental evaluation impose a tricky manual instrumentation of the existing code. CatchIt is intended to fill these limitations using work domain descriptions as the central foundation.

In the next section, we address the state of the art on approaches to usability using work domain descriptions and motivate the requirements for CatchIt. We then present the principles of CatchIt in details and close the discussion with the perspectives for this work.

STATE OF THE ART

Usability testing of complex systems requires computer support to fit the temporal and financial constraints. Unfortunately, current commercial products for evaluating user interfaces are too limited in scope to be relevant for critical systems. Within the research community, pro-active and reactive tools have been developed to address usability issues: pro-active tools aim at supporting the construction of high quality user interfaces whereas reactive tools are concerned with the evaluation of interactive systems.

Within the pro-active approach, model-based interface builders have demonstrated the power of work domain descriptions. They rely on declarative specifications of the semantics of the domain as well as on any source of knowledge useful for generating the rendering and the dynamic behavior of the target system. Model-based interface builders gather as much work domain knowledge as possible, and aim at identifying reusable software components. By doing so, they minimize the amount of procedural code that needs to be implemented [35]. UIDE [12], ADEPT [19], MECANO [31], FUSE [23], AME [25], HUMANOID [37] and MASTERMIND [38] have demonstrated the technical feasibility of the approach.

Within the reactive approach, some tools evaluate system use while others support evaluation. The first category of tools, such as OPADE [9], FRAMER [22], CRACK [10] and JANUS-CRACK [11], provides the end-user with explanations about the appropriate way of using the system. They do not fit our concerns. Tools for evaluating a particular design may be predictive or experimental:

- Predictive evaluation techniques allow the detection of usability problems without testing the system with users. Predictive approaches may be based on a theory: the GOMS [7] and TAG families [30], PUM [41] and ICS [4], are good examples of theory-based evaluation methods. Alternatively, HCI heuristics may be used to evaluate a user interface without user testing [28]. Computer tools that best illustrate the predictive approach include GLEAN [20] a computerized version of GOMS, KRI [24], SYNOP [21], CritiGUI [29], AIDE [33] and USAGE [5].
- Experimental approaches, on the other hand, involve end users participation. Typical illustrations of experimental approaches include: interviewing the user, observing and monitoring users' behavior using various supports (e.g., usability lab, paper and pencil, wizard of Oz apparatus) combined to multiple complementary techniques (e.g., measurement tests, thinking aloud, etc.). Computer-based tools for experimental evaluation vary widely in terms of their functional coverage. For example, Playback [27], NEIMO [1, 8] and Hammontree's system [17] favor the electronic recording of behavioral data; MRP [34] and EMA [2] cover the analysis of behavioral data and VDDE [36] is targeted at explanations for phone-based interaction.

The analysis of the state of the art calls for the following remarks:

- First, automated tools adhere to a single paradigm: construction vs. evaluation. Although the UIDE interface builder attempts to include GOMS-based evaluation, it supports predictive evaluation at the keystroke level only. In particular, it does not include the automatic monitoring of users behavior to feed into experimental evaluations.
- The automated monitoring of behavioral data requires the source code of the user interface to be instrumented. This programming task can be avoided by using a modified instrumented version of widgets classes as those provided in current user interface toolkits (Cf. Hammontree's approach [17]). This technique fails at tracking user's actions on work domain objects that are not implemented as widgets. It does not either cover events at high level of abstraction such as the beginning or the end of a task. Our experience with NEIMO [8] shows that instrumenting the code by hand at various levels of abstraction is not only a time consuming task but a source of programming errors.
- The output of an automated evaluation tool may range from the replay of behavioral data to the automatic correction of design errors. While replay is at too low a level of abstraction and forces humans to perform a heavy evaluation task, auto-correction is still unrealistic (and probably not desirable) [2,1]. Detection and explanation of anomalies is a reasonable medium range goal to consider.

CatchIt aims at filling some of the above limitations while exploiting successful current approaches and mechanisms.

CATCHIT

CatchIt is intended to cover both the development and the evaluation process of interactive systems. With regard to evaluation, it aims at supporting the predictive and experimental approaches to usability testing. Work domain descriptions provide the foundations for these goals. As discussed above, they serve as a central mechanism for traceability and reusability within a particular work domain. In the next sections, we present an overview of the functional coverage of CatchIt followed by a detailed

description of each of its components. The approach is illustrated on a military application provided by THOMSON-CSF Detexis. For reasons of confidentiality and pedagogy, the case study is reduced to a single user task: the monitoring of a tactical situation. A tactical situation provides information about the surrounding environment: for example, the position and speed of objects (tracks) that have been detected by the embedded equipments. In case of danger, the operator has to deploy the right tactic. A maintenance mode is available to maintain equipments. The CatchIt facilities as well as their limitations in the current Smalltalk [13] implementation are illustrated on this example.

Functional Coverage

As shown in Figure 2, an executable work domain model forms the heart of the CatchIt environment. This kernel, which can be generated from high level specifications, defines a reference model. This reference model serves two complementary purposes: it can be used as reusable code from which a new application can be built using a browser or it can be used as a standard against which new applications can be tested. Evaluation, whether it be predictive or experimental, consists of the automated detection of deviations of the new application from the norm defined by the reference model.

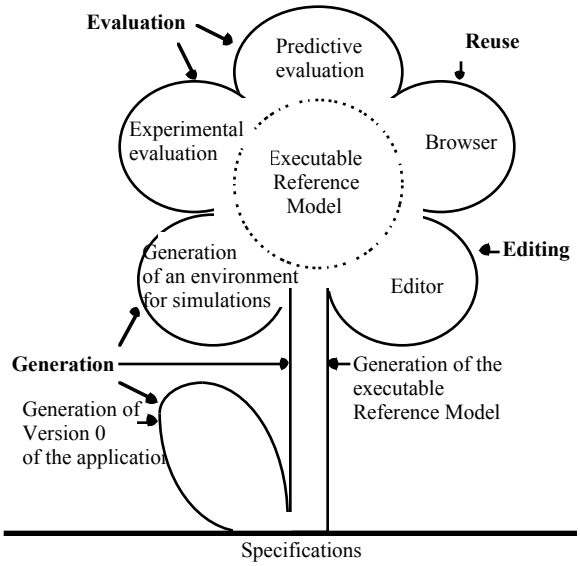

Figure 2. The functional coverage of CatchIt.

Experimental testing requires two extra mechanisms: code instrumentation at various levels of abstraction and realistic simulation of the operational context. Because code instrumentation is a time consuming and error prone process, it is generated automatically by CatchIt in a transparent way. Figure 3 shows the principles: just like an electro-cardiogram, CatchIt probes the application without requiring programmers to modify their code manually. Similarly, a simulation environment can be generated to control the operational coverage of the tests.

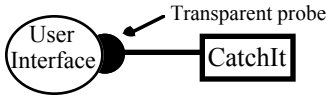

Figure 3. CatchIt as a probe for instrumenting the application to be evaluated.

The reference model evolves over time based on the accumulation of past experience. A dedicated editor can be used to update the model accordingly.

The Reference Model

The reference model is an executable description of a particular work domain. It aims at making explicit information that programmers typically embed in an implicit manner within the application code.

The CatchIt reference model covers three classes of domain knowledge: domain concepts, operator's tasks, and operator's strategies. Whereas a task prescribes the procedures to reach a goal, a strategy expresses the conditions under which a task or a set of tasks is performed. Clearly, preconditions supported in formalisms like MAD [32] encompass the notion of strategies within the task description. However, the variability of strategies across contexts triggers the need for an explicit concept distinct from that of a task. The following example shows a simplified version of a reference model for monitoring a tactical situation.

Domain concepts

> *TacticalSituation*: tracks, equipments ...
>
> *Track*: identification, position, speed, course, criticity ...
>
> *Equipment*: identification, state ...
>
> *Mode*: state (operational vs. maintenance)

Tasks

> *MonitorTacticalSituation*:
>
> - precondition: -
>
> - procedure: *MonitorTracks* and *MonitorEquipments*
>
> - postcondition: -
>
> *MonitorTracks: ...*
>
> *MonitorEquipments: ...*
>
> *InvigilateATrack:*
>
> - precondition: -
>
> - procedure: *EvaluateCriticity* then *ChooseTactic* then *DeployTactic*
>
> - postcondition: *theTrack isNotCritic*
>
> *SwitchToMaintenanceMode*: ...
>
> *SwitchToOperationalMode: ...*

Strategies

IF *a new track is detected* THEN *invigilate that track* is MANDATORY

IF *a track is critic* THEN *switch to maintenance mode* is UNDESIRABLE

Figure 4 shows the main panel for specifying the concepts, tasks and strategies of a reference model. In CatchIt environment, a reference model can be viewed as a specialization of a more general reference model. For example as shown in Figure 4, the naval electronic warfare (Guerre Electronique Marine) is defined as a specialization of electronic warfare (Guerre Electronique).

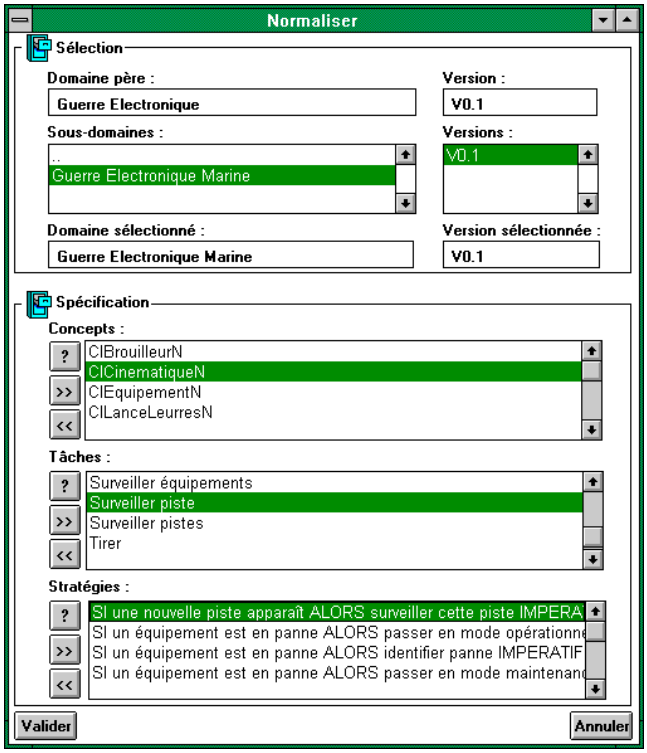

Figure 4. The CatchIt environment for defining reference models.

Whereas concepts are defined as object oriented classes within the Smalltalk browser, tasks and strategies are specified in dedicated panels. Figure 5 shows the form for specifying a task. It is illustrated on the task *Invigilate a track*. This task is decomposed into three subtasks : *EvaluateCriticity* (Evaluer criticité) then *ChooseTactic* (Choisir tactique) then *DeployTactic* (Déployer tactique).

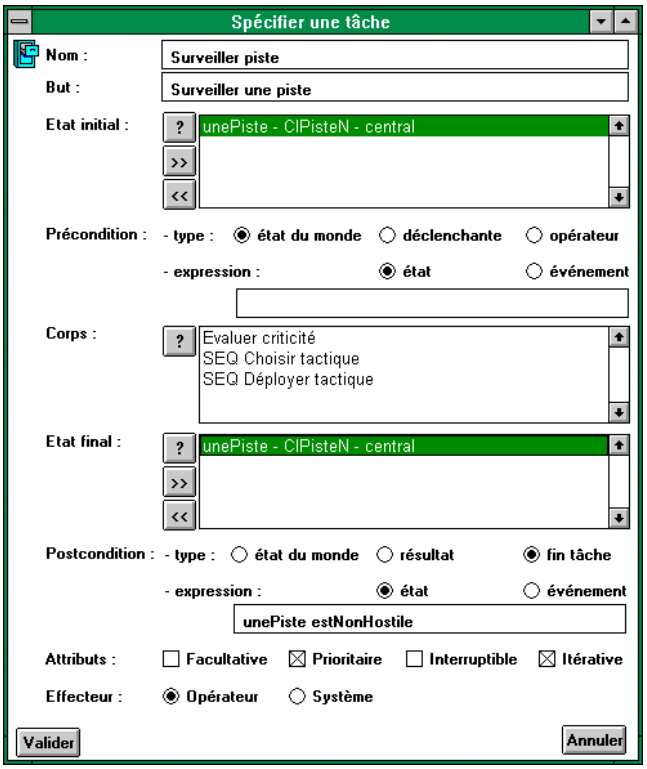

Figure 5. The CatchIt environment for specifying a task.

Figure 6 shows the panel for specifying a strategy. It is illustrated on the strategy "IF *a new track is detected* (une nouvelle piste apparaît) THEN *invigilate that track* (surveiller cette piste) is MANDATORY (impératif)".

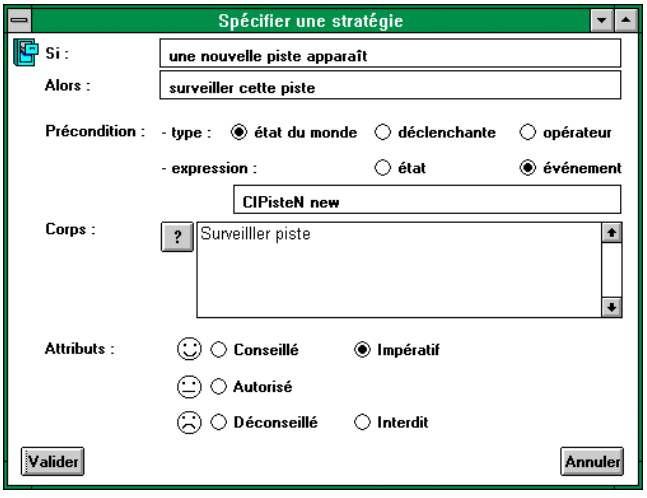

Figure 6. The form for specifying a strategy.

As shown in Figures 5 and 6, pre- and post-conditions within tasks as well as predicates within strategies are grounded on instruction blocks expressed in the context of the reference model. For example in Figure 5, the post-condition "*theTrack isNotCritic*" is expressed by the Smalltalk instruction "*unePiste estNonHostile*". In Figure

6, the condition "*a new track is detected"* is expressed by the Smalltalk expression "*CIPisteN new*".

Pre- and post-conditions within tasks as well as predicates within strategies are evaluated either in the context of the reference model or within the new application. This duality is supported by the specification of explicit links between the reference model and the application.

Connecting the Application with the Reference Model

Connections between the application and the reference model are specified by the application developer. This declarative mapping is the only task that the developer needs to perform in order to use CatchIt. The specification gives birth to a dedicated software component, the adaptor. This adaptor, which maintains the semantic links between the application and the reference model, acts as a connector. By doing so, connection is performed without polluting the application nor the reference model with extra code.

Connecting the application to the reference model is structured according to the three classes of entities supported by CatchIt: concepts, tasks and strategies. The mapping consists in linking the entities of the reference model to entities of the application in terms of abstraction and presentation. For example, in terms of abstraction, as the concept of mobile tactical object does not explicitly exist in the application, the class *CIObjetTactiqueMobileN* of the reference model is linked to the class *CIObjetTactiqueA* of the application. As shown in figure 7, this mapping is refined at the level of attributes. The field *cinematic* (cinématique) of *CIObjetTactiqueMobileN* is connected with the field *speed* (vitesse) of the class *CIObjetTactiqueA*.

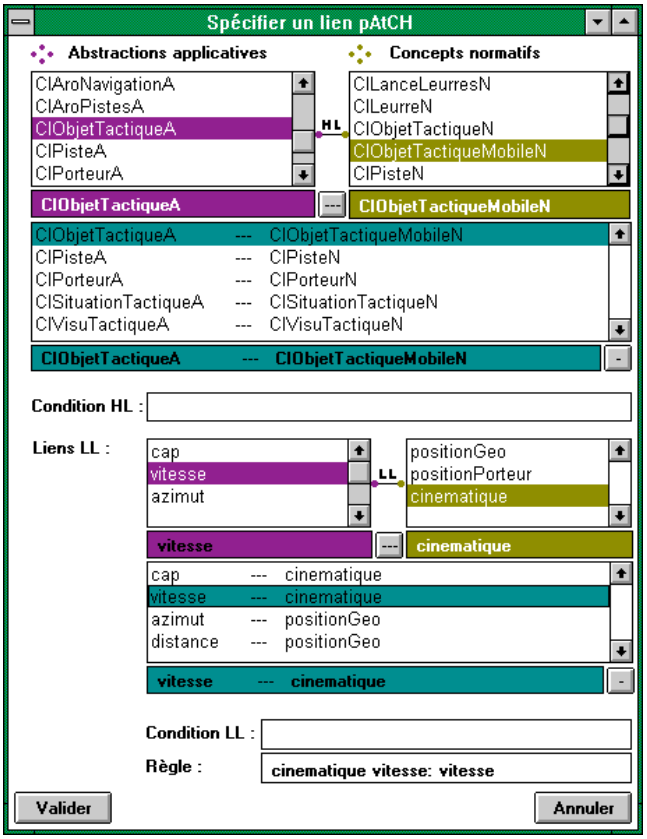

Figure 7. The form for linking concepts of the reference model to abstractions in the application.

Presentation entities of the application such as widgets, working spaces, or a combination of user's actions on input devices are mapped to a task of the reference model when they correspond to a task, or to a concept of the reference model when these entities represent a concept. For example, the class *CIAroModeFctA* of the application that contains the radio buttons in charge of the mode switching is linked to the task *SwitchToMaintenanceMode* (passer en mode maintenance) of the reference model. The expression of these links by the developer is performed within CatchIt using form-based interaction (Figure 8).

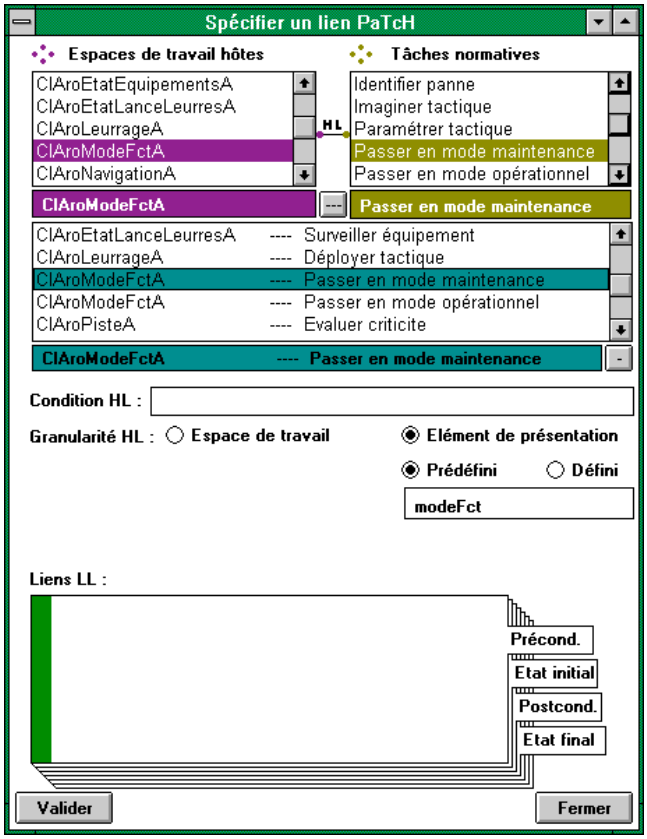

Figure 8. The form for linking tasks of the reference model to presentations in the application.

Each mapping is grounded on an instruction block that connects together the software of the application and the reference model. This set of instructions is modelled as a "proxy" method within the adaptor. For example in Figure 7, the cinematic field of the reference model is updated according to the Smalltalk mapping rule: *cinematique vitesse: vitesse*.

Typically predicates that express properties related to rendering such as observability, are evaluated within the context of the application. Conversely, predicates that address states about domain objects are evaluated within the context of the reference model. Examples will be provided in further sections to illustrate the power of this facility.

In summary, a dedicated component, the adaptor, gathers the semantic links between the application and the reference model that are specified declaratively by the developer. Based on the knowledge of these links, a predictive evaluation can be performed.

Predictive Evaluation

CatchIt supports a predictive static evaluation based on the completeness and correctness of the links. Any anomaly is reported to the designer as warnings. Typically, CatchIt checks that any relevant data in the reference model has its counterpart in the application. It also tests the quality of the application code by measuring the number of non-bijective links. For example, CatchIt can detect that the classes *ReferenceApplication* and *TacticalSituation* of the reference model are both connected to the class *Application* of the application. In other words, CatchIt checks the balance between the application and the reference. Finally, CatchIt can predict the operator's overload in a given context by counting the number of strategies that cannot migrate to the system.

As of this writing, the predictive evaluation has been partially implemented. It is currently limited to the verification of links completeness. Instead, we have focussed our attention on experimental evaluation.

Experimental evaluation

CatchIt uses the links specified by the developer to automatically instrument the source code of the application. The code generated by CatchIt results in the automatic propagation of relevant events towards the reference model. This notification mechanism sets the foundations for the experimental evaluation of the application. Code instrumentation and the exploitation of events notification are presented next.

Propagation and Transparence

Every method in the application that performs a write access to a linked object is automatically instrumented. The instrumentation consists of inserting a spy instruction that redirects an event to the adaptor (Figure 9). The insertion occurs as the last instruction of the method. The method is recompiled but the spy instruction is not observable to developers even when browsing the source code. This is our concept of transparence.

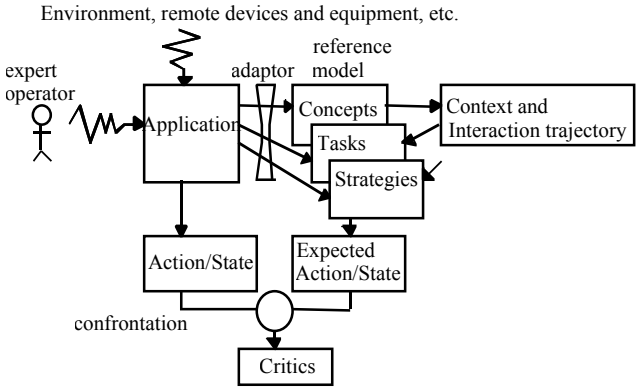

Figure 9. Principles of the experimental evaluation in CatchIt.

The CatchIt transparent code instrumentation allows every modification of a linked application object to be echoed within the reference model (Figure 9). Echoing is made possible through the proxies maintained by the adaptor. These proxies, which are linked to application objects, are instances of classes defined in the reference model. Thus, any modification on an application object is notified to the adaptor which invokes the appropriate methods on proxies. This set of proxies and their state across time define the

interactional context. The history of end-user's actions defines the *interactional trajectory*. (At the time of this writing, the concept of interactional trajectory is partially implemented.) The notion of interactional context coupled with that of interactional trajectory makes it possible to compare the operator's intention with the manifestation of the intention. This comparison is at the basis of our evaluation technique.

Principles of the evaluation technique

As shown in Figure 9, every event, whether it comes from a remote device such as a sensor or from the operator when acting on application objects, is processed by the application. In addition, when linked application objects are modified, the event is automatically propagated to the adaptor by the transparent instrumentation mechanism. In turn, the notification results in the modification of the interactional context and trajectory.

The state of the interactional context and trajectory allows CatchIt to identify the operator's intention. The operator, i.e., a representative subject involved in the experimental evaluation of the application, is supposed to be an expert. This decision, which has been viewed as a shortcoming for GOMS, is a reasonable choice for the work domain we are interested in: by definition, users of critical systems are experts. Since, by definition, the reference model defines THE best developed solution for expert users, there should be a direct mapping between the operator's intention and its reflection in the state of the reference model whose relevant portion is modelled as the interactional context and trajectory.

Therefore, the job of the critics is to compare the operator's action and application state with the expected action and state maintained in the interactional context and trajectory.

The critics

The critics are based on the correctness, completeness and conciseness of the user behavior (Figure 10). Whereas the correctness checks that any task performed by the operator is appropriate, the completeness checks that any expected task is really performed. The conciseness measures to which extent only mandatory tasks are performed.

From [10], we select two orthogonal dimensions for characterizing the output of critics (Figure 10): the publication mode (i.e., on the fly vs. off line) and the polarity (i.e., positive vs. negative). Whereas on the fly notification is disruptive for the subject, an off line mechanism builds a log of warnings that can be analyzed after the experiments. Positive polarity denotes conformity to the reference while negative polarity points out deviations from the reference.

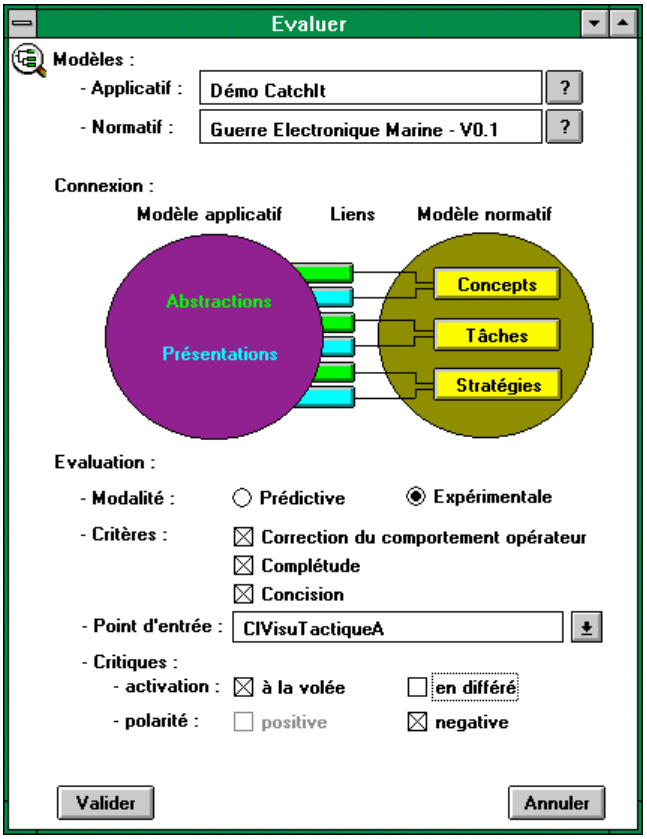

Figure 10. The panel for launching an experimental evaluation.

In the current implementation, the CatchIt critics lists the deviations between the expected and perceived behaviors of the application as well as possible explanations of the deviations. For example, if although a track is critic, the operator switches to maintenance mode, CatchIt detects the transgression of a strategy (*IF a track is critic THEN switch to maintenance mode is UNDESIRABLE*) and suggests the non observability of either the track or the mode. Figure 11 shows this example of negative critic published on the fly.

In the future, the critics could check that a mapping between the concept *Track* in the reference model has been linked to a presentation item in the application. If so, it could check that this presentation item is effectively observable: to do so, it could check that a display statement, tracked automatically by the instrumentation process, has been performed in the application code.

Correction

CatchIt detects deviations but do not perform any corrective actions. This task falls to the developer. However, for tasks and strategies that do not conform to the expected behavior, CatchIt could improve its contribution by providing an ordered list of possible causes based on the concept of variability. Typically, for a particular domain, concepts are more stable than tasks which in turn are less variable than strategies. Therefore, if CatchIt detected a conflict between an object and a strategy, it would point out the strategy

first. This is a very speculative hypothesis that we need to investigate further.

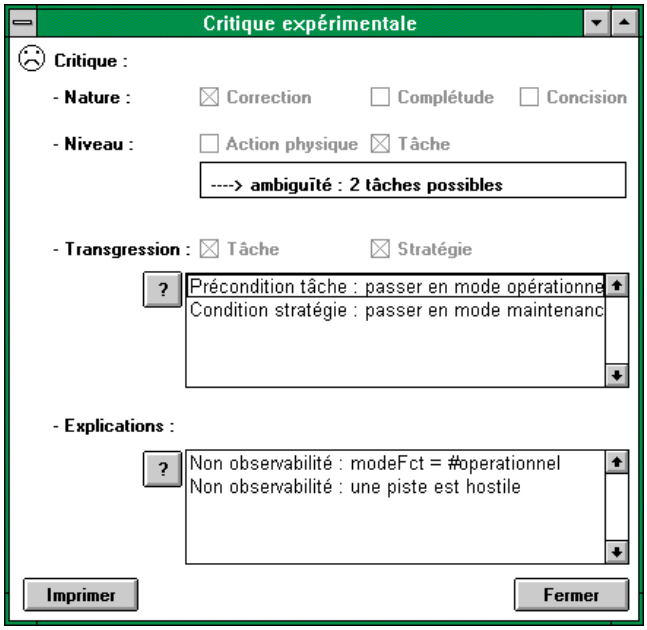

Figure 11. The panel for publishing an experimental critic on the fly.

Simulation

Human resources such as subjects, are scarce and expensive. As a result, scenarios used in experiments generally focus on specific aspects of the system. In the CatchIt environment, the purpose of the simulator is to generate events that improve the operational coverage of experimental tests. Typically, CatchIt builds a set of events from the task preconditions as well as from the conditions that trigger the strategies of the reference model. CatchIt can simulate the occurrence of such events within the application. By so doing, it provides another way of validating an application against the reference model without using effective users.

Browsing and Editing

In its current form, CatchIt reuses the browser and the editor provided by the Smalltalk environment. As mentionned above, these facilities are used by developers to build new applications or to elaborate a new version of the reference model.

Generating Reference Models and Applications

Building new applications or elaborating a reference model for a particular work domain can be built by hand using programming tools such as Smalltalk. Alternatively, applications and reference models can be generated automatically using a dedicated specification language. In the current stage of its development process, CatchIt does not offer any formalism to alleviate the programming task. The definition of an appropriate formalism is one of the action tasks of our research agenda.

LIMITATIONS AND FUTURE WORK

Although we have demonstrated the technical feasibility of transparent instrumentation as well as the articulation of predictive evaluation with experimental testing, we have not explored yet the benefits and limitations of our approach. In particular, we have not integrated all aspects of properties such as insistance, honesty, reachability, recoverability, and task migration [40]. We need to define metrics for these properties in order to refine the deviations detected by the critics. A model of deviations should be developed to enhance the expressive power of the critics. Finally, we need to verify that the reference model is rich enough to capture knowledge structures [39] and investigate its extension for supporting multi-user interactive systems.

The technical feasibility of our approach imposes two pre-requisites: first, the application and the executable reference model are implemented using an object-oriented language; second, CatchIt needs to have access to the source code in order to perform the instrumentation. With regard to the first requirement, the industrial trend is the use of object-oriented languages. As for the second problem, programming by components as examplified by ActiveX and OLE [42], makes it possible to interoperate CatchIt with applications to be tested. In addition, we are currently evaluating whether the AOP approach (Aspect Oriented Programming) is appropriate for the implementation of the probe.

CONCLUSION

Although CatchIt is still an experimental platform, its current implementation has demonstrated the technical feasibility of the approach to the development and evaluation of realistic applications.

CatchIt offers a new way of approaching the software development of interactive software through the capitalization of work domain knowledge across time. So far, the work domain covers the notions of domain concepts, tasks and strategies. The executable form of this knowledge defines a reference model from which new applications can be elaborated and against which new applications can be evaluated. Predictive and experimental evaluations can be used in complementary ways. Both of them are grounded on the declarative expression of connections between items of the reference model and their corresponding elements in the application. These specifications, which are performed by the developer using a form-based user interface, are used by CatchIt to instrument the source code in a transparent way. This transparency feature alleviates a time consuming and error prone task.

Early experience with the use of CatchIt shows that developers, guided by the reference model, tend to improve the structure of their applications. Besides defining the right programming patterns and a platform for evaluation, an environment like CatchIt can be extended to other uses such as training operators and computer-supported assistance.

ACKNOWLEDGMENTS

We thank THOMSON-CSF Detexis for having supported this work.

REFERENCES

1. Aublet-Cuvelier, L., Carraux, E., Coutaz, J., Nigay, L., Portolan, N., Salber, D., Zanello, M.L. NEIMO, un laboratoire d'utilisabilité numérique : Leçons de l'expérience. *ERGO IA 96*, pp 149-160.
2. Balbo, S. *Evaluation Ergonomique des Interfaces Utilisateur : Un Pas Vers l'Automatisation*. Thèse de l'Université Joseph Fourier de Grenoble, September 1994, 287 p.
3. Bares, M., Pastor, D. Principe d'un moteur d'interaction multimodale pour systèmes embarqués. *Génie Logiciel*, N°40, Juin 1996, 31-38.
4. Barnard, P. Cognitive Resources and the Learning of Human-Computer Dialogs, in *Interfacing Thought*, J.M. Carroll eds, The MIT Press, 1987, pp. 112-158.
5. Byrne, M.D., Wood, S.D., Sukaviriya, P.N., Foley, J.D., Kieras, D.E. Automating Interface Evaluation. *CHI'94*, Boston, April 1994, pp 232-237.
6. Calvary, G.: Proactivité et réactivité: de l'Assignation à la Complémentarité en Conception et Evaluation d'Interfaces Homme-Machine, *Phd of the University Joseph-Fourier-Grenoble I-France,* Speciality Computer Science, 1998, 250 p.
7. Card, S., Moran, T., Newell, A. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
8. Coutaz, J., Salber, D. and Carraux, E. NEIMO, a Multiworkstation Usability Lab for Observing and Analyzing Multimodal Interaction. In Proc. CHI96 conference companion, ACM publ., Tauber, M. Ed., 1996, pp. 402-403.
9. De Rosis, F., Cozza, M.T., De Carolis, B., Errore, S., Pizzutilo, S., De Zegher, I. Adaptative Interaction With Knowledge-Based Systems. *AVI'94*, Bari, Italy, June 1-4, 1994.
10. Fischer, G., Morch, A. CRACK : A critiquing approach to cooperative kitchen design. *Proceedings of the ACM International Conference on Intelligent Tutoring Systems*, pp 176-185, May 1988.
11. Fischer, G., Lemke, C., Mastaglio, T., Morch, A.I. Using Critics to Empower Users. *CHI'90*, April, 1990, pp 337-347.
12. Foley, J., Kim, W.C., Kovacevic, S., Murray, K. *The User Interface Design Environment*. GWU-IIST-88-04, Department of Electrical Engineering and Computer Science, Scool of Engineering and Applied Science, The George Washington University, Washington, D.C. 20052, January 88.
13. Goldberg, A. *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley, 1984.
14. Gould, J.D. How to Design Usable Systems, *Handbook of Human-Computer Interaction*. M. Helander ed., Elsevier Science B.V., 1988, pp 757-789.
15. Goodman, S.E. War, Information Technologies, and International Asymmetries. *Communications of the ACM*, December 1996, Vol. 39, Number 12, pp 11- 15.
16. Hall, A. Do Interactive Systems Need Specifications? In Proc. 4th Eurographics Workshop on DSVIS'97, pp. 3-14.
17. Hammontree, M.L., Hendrickson, J.J., Hensley, B.W. Integrated data capture and analysis tools for research and testing on graphical user interfaces. *CHI'92*, Monterey, 3-7 May 1992, pp 431-432.
18. Holyer, A. Methods For Evaluating User Interfaces, *Cognitive Science Research Paper No. 301*, Nov. 1993.
19. Johnson, P., Wilson, S., Markopoulos, P., Pycock, J. ADEPT - Advanced Design Environment for Prototyping with Task Models. *InterCHI'93*, 24-29 April, 1993, pp 56-57.
20. Kieras, D.E., Wood, S.D., Abotel, K., Hornof, A. GLEAN : A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs. *UIST'95*, Pittsburgh, Pennsylvania, November 14-17, 1995, pp 91-100.
21. Kolski, C. *Contribution à l'ergonomie de conception des interfaces graphiques homme-machine dans les procédés industriels : application au système expert SYNOP*. Thèse de l'Université de Valenciennes et du Hainaut-Cambrésis, Janvier 1989.
22. Lemke, A., Fischer, G. A cooperative problem solving system for user interface design. *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, pp 479-484.
23. Lonczewski, F., Schreiber, S. The FUSE-System : an Integrated User Interface Design Environment, *CADUI'96*, J. Vanderdonckt (eds), 1996, pp 37-56.
24. Löwgren, J. A Knowledge-Based Tool for User Interface Evaluation and its Integration in a UIMS. *Interact'90*, 1990 , pp 395-400.
25. Märtin, C. Software Life Cycle Automation for Interactive Applications : The AME Design Environment. *CADUI'96*, J. Vanderdonckt (eds), 1996, pp. 57-73.
26. Moran, T. and Carroll, J. *Design Rationale, Concepts, Techniques and Use*, Lawrence Erlbaum Publ., 1996.
27. Neal, A.S., Simons, R.M. Playback : a method for evaluating the usability of software and its documentation. *CHI'83*, December 1983, pp 78-82.
28. Nielsen, J. and Molich, R. Heuristic evaluation of user interfaces, Proceedings of the CHI'90 Conference on Computer Human Interaction, Seatle, ACM New York, 1990, pp. 349-256.
29. Nitsche-Ruhland, D. Zimmermann, G. CritiGUI-Knowledge-based Support fort the Interface Design

Process in Smalltalk. *EWHCI'95*, Moscow, Russia, July 1995.

30. Payne, S.J. and Green, T.R.G. Task-Action Grammars: a model of the mental representation of task languages. In Human-Computer Interaction, Vol. 2, No. 2, Lawrence Erlbaum Associates Publishers, 1996, pp. 93-133,.
31. Puerta, A. The MECANO Project : Comprehensive and Integrated Support for Model-Based Interface Development. *CADUI'96*, J. Vanderdonckt (eds), 1996, pp 19-35.
32. Scapin, D.L., Pierret-Golbreich, C. MAD : Une méthode analytique de description des tâches. *IHM'89* , Sophia-Antipolis, Mai 1989, pp 131-148.
33. Sears, A. AIDE : A step toward metric-based interface development tools. *UIST'95*, November 14-17, 1995, pp 101-110.
34. Siochi, A.C., Hix, D. A study of computer-supported user interface evaluation using maximal repeating pattern analysis. *CHI'91*, New Orleans, May, 1991, ACM New-York, pp 301-305.
35. Sukaviriya, N., Kovacevic, S., Foley, J.D., Meyers, B.A., Olsen, D.R., Schneider-Hufschmidt, M. Model-Based Interfaces, What are They and Why Should We Care ?. *UIST'94*, November 2-4, 1994, pp 133-135.
36. Summer, T., Bonnardel, N., Kallak, B.H. The Cognitive Ergonomics of Knowledge-Based Design Support Systems. *CHI'97*, 22-27 march 1997, Atlanta, Georgia, pp 83-90.
37. Szekely, P., Luo, P., Neches, R. Facilitating the Exploration of Interface Design Alternatives : The HUMANOID Model of Interface Design. *CHI'92*, May 3-7, 1992, Monterey, California, pp 507-515.
38. Szekely, P., Sukaviriya, P., Castells , P., Muthukumarasamy, J., Salcher, E. Declarative interface models for user interface construction tools: the MASTERMIND approach. *EHCI'95*.
39. Waern, Y., Hägglund, S., Ramberg, R., Rankin, I., Harrius, J. Computational advice and explanations - behavioural and computational aspects. *Interact'95*, pp 203-206.
40. WG 2.7 (13.4). *Design Principles for Interactive Software*, Gram, C. & Cockton, G., Eds., Chapman&Hall Publ., 1996.
41. Young, R.M. and Whittington, J. A knowledge Analysis of interactivity, Proceedings of INTERACT'90, D. Diaper, G. Cockton & B. Shackel Eds., Elsevier Scientific Publishers B.V, 1990, pp. 207-212.
42. Chappel, D. *Understanding ActiveX and OLE, a Guide for Developers and Managers*, Microsoft Press, 1996.