



A Software Tool for the Design of Multifunction Displays

By

Gregory Francis

Purdue University

Aircrew Health and Performance Division

September 1999

Approved for public release; distribution unlimited.

**U.S. Army Aeromedical Research Laboratory
Fort Rucker, Alabama 36362-0577**

Notice

Qualified requesters

Qualified requesters may obtain copies from the Defense Technical Information Center (DTIC), Cameron Station, Alexandria, Virginia 22314. Orders will be expedited if placed through the librarian or other person designated to request documents from DTIC.

Change of address

Organizations receiving reports from the U.S. Army Aeromedical Research Laboratory on automatic mailing lists should confirm correct address when corresponding about laboratory reports.

Disposition

Destroy this document when it is no longer needed. Do not return it to the originator.

Disclaimer

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation. Citation of trade names in this report does not constitute an official Department of the Army endorsement or approval of the use of such commercial items.

REPORT DOCUMENTATION PAGE				<i>Form Approved OMB No. 0704-0188</i>	
1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release, distribution unlimited			
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) USAARL Report No. 99-20		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION U.S. Army Aeromedical Research Laboratory		6b. OFFICE SYMBOL <i>(If applicable)</i> MCMR-UAS	7a. NAME OF MONITORING ORGANIZATION U.S. Army Medical Research and Materiel Command		
6c. ADDRESS <i>(City, State, and ZIP Code)</i> P.O. Box 620577 Fort Rucker, AL 36362-0577		7b. ADDRESS <i>(City, State, and ZIP Code)</i> Fort Detrick Frederick, MD 21702-5012			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL <i>(If applicable)</i>	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS <i>(City, State, and ZIP Code)</i>		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO. 62787A	PROJECT NO. 30162787A879	TASK NO. PB	WORK UNIT ACCESSION NO. DA336445
11. TITLE <i>(Include Security Classification)</i> A software tool for the design of multifunction displays. (U)					
12. PERSONAL AUTHOR(S) Gregory Francis					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO	14. DATE OF REPORT <i>(Year, Month, Day)</i> 1999 September		15. PAGE COUNT 65
16. SUPPLEMENTAL NOTATION					
17. COSATI CODES			18. SUBJECT TERMS <i>(Continue on reverse if necessary and identify by block number)</i> cockpit design, hierarchy, multifunction displays, workload		
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT <i>(Continue on reverse if necessary and identify by block number)</i> This document describes MFDDTool, a software aid for the design of multifunction displays (MFDs). MFDDTool applies an optimization algorithm to designer-specified constraints thereby creating the best layout of MFD information for MFD hardware. The guide specifies the types of MFD situations where MFDDTool applies and describes the steps needed to define constraints and start the optimization approach. A sample MFD design problem (involving an automated teller machine) is discussed. Appendices include both source code to the software and a user's guide to MFDDTool.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Chief, Science Support Center		22b. TELEPHONE <i>(Include Area Code)</i> (334) 255-6907		22c. OFFICE SYMBOL MCMR-UAX-SI	

Table of contents

	<u>Page</u>
Introduction.....	1
MFDTool	2
Constraint costs	3
Weighting costs	6
Optimization	7
Conclusions.....	9
References.....	10
Appendix A. Java source code.....	12
Appendix B. MFDTool user's guide.....	36

Introduction

Previous research (Francis & Reardon, 1997; Francis, 1998) described a computational method of assigning labels and functions to user selections in a multifunction display (MFD). An MFD consists of a computer display screen that provides information in response to a user's request. Through interaction with an interface (usually button pushes), the user moves through a hierarchical representation of information. Automated teller machines, medical devices, aircraft cockpits, electric typewriters, retail registers, fax machines, and many other devices utilize MFDs, with varying degrees of complexity.

At some point in the design of an MFD, decisions must be made about how to map the various parts of the information hierarchy to user actions (e.g., button pushes). These decisions can affect user performance, as demonstrated by Rogers et al. (1996) for automated tellers, Obradovich and Woods (1996) and Cook and Woods (1996) for medical devices, Cuomo et al. (1998) for intranet information servers, and Reising and Curry (1987), Dohme (1995), and Sirevaag et al. (1993) for aircraft flight.

Mapping hierarchy information to MFD buttons is a challenging task. The human-computer interactions involved in accessing information from an MFD are complicated and not entirely understood. Moreover, even a small hierarchy database can be mapped to hardware buttons in a vast number of ways (see Fisher, Yungkurth and Moss (1990) for a discussion of this issue), so combinatorial explosion quickly precludes an exhaustive search of all possible mappings. Therefore such mappings are, at best, created by experts who rely on experience and general guidelines (Calhoun, 1978; Lind, 1981; Spiger and Farrell, 1982; MIL-STD-1472D; Williges, Williges and Fainter, 1988; Holley and Busbridge, 1995). Francis and Reardon (1997) summarized many of these guidelines:

1. Frequently used functions should be the most accessible.
2. Time critical functions should be the most accessible.
3. Frequently used and time critical functions should be activated by the buttons that feel "ideally located" (e.g., top of a column of buttons).
4. Program repeated selection of the same button. For example, locate the most commonly selected function of a menu on the same button that called up that menu.
5. The number of levels in the hierarchy should be as small as possible.
6. The overall time to reach functions should be minimized.
7. Functions that are used together should be grouped on the same or adjacent pages.
8. Related functions on separate pages should be in a consistent location.
9. Related functions should be listed next to each other when on a single page.
10. Consider the types of errors crew members might make and place functions accordingly to minimize the effect of those errors.

Many of these guidelines are the same as those applied to the physical layout of controls (Sanders and McCormick, 1987), while others (4, 5, 8, 9) appear to be unique to the search of an electronic database. Some of these guidelines have been investigated experimentally. For example, Snowberry, Parkinson and Sisson (1983) showed that hierarchy search speed and accuracy increased as the number of levels decreased (5). Likewise, Teitlebaum and Granda (1983) demonstrated that placing related functions in inconsistent positions resulted in a 73% increase in

search time (8). A literature search found no experimental evidence to support the remaining guidelines, although many of them seem reasonable.

While many of these guidelines correctly identify the key characteristics of good MFD design, application of these criteria is problematic because they often conflict with each other. For example, should a frequently used function be placed by itself near the top of the hierarchy (1) or should it be placed in a submenu with its related, but infrequently used, functions (7)? Likewise, should criteria (3), (4) or (7) dominate selection of a button for a specific function? Currently, there is no quantitative method of measuring the tradeoffs and designers try out different options until the whole system “feels” good. This is a time consuming task because movement of a single label can require additional changes throughout the MFD. As a result, MFD design largely remains an artistic endeavor, depending primarily on the experience, intuition, and hard work of the designer.

Recently, there has been interest in computational methods that can optimize the design of computer consoles (Roske-Hofstrand and Paap, 1986; Fisher, 1993; Hwa, Marks and Shieber, 1995; Farrell, 1997; Sargent, Kay and Sargent, 1997). The general approach is to gather relevant data about the human computer interaction, build a quantitative model of that interaction, and then find a design that provides the best model performance. Our previous research (Francis and Reardon, 1997; Francis, 1998) demonstrated how to apply this approach to assigning MFD labels to button presses. While successful, the applications were tailored to the specific task at hand. To broaden the applicability and use of the approach, we created a software tool that can be used by MFD designers in more general situations to optimize the association of MFD labels to buttons.

MFDDTool

MFDDTool is computer software that provides a graphical interface for an MFD designer to build an MFD hierarchy and associate the hierarchical information with physical buttons on specified MFD hardware. Through the graphical interface, the designer creates a variety of constraints on the MFD and the program applies a variation of the algorithms described in Francis and Reardon (1997) and Francis (1998) to identify the best MFD design that satisfies the designer-specified constraints.

MFDDTool focuses on a subset of the guidelines identified above that can be recast in terms of an optimization problem. MFDDTool requires that the designer has a specified MFD hardware system that describes the sizes and positions of MFD buttons (the approach can be modified to other types of interactions, but buttons are a common interface type). MFDDTool also requires that the designer specifies the hierarchical arrangement of information pages in the database. This arrangement also allows for hyperlinks that move back up the hierarchy (e.g., *RETURN*) or function as shortcuts.

Given this information, MFDDTool allows the user to identify four types of constraints, which can be mixed and matched as desired.

- **Global movement time:** If one ignores shortcuts and backward links, then moving through the hierarchy from the top to any specified page can be described by a single finite sequence

of button presses. When the designer specifies the frequencies of search for different pieces of MFD information, MFDTool associates page labels with buttons in a way to minimize the average movement time needed to reach information. This constraint corresponds to guidelines (1), (6), and often (4), above.

- **Pages to close buttons:** Often labels on a single screen are related to each other and the designer wants the related page labels to be grouped together on nearby buttons. At other times labels on different MFD screens are related and the designer wants those labels to be associated to the same or nearby buttons (e.g., *CANCEL* should be in the same place on every page). MFDTool allows the designer to specify as many of these constraints as desired. This constraint corresponds to guidelines (8), and (9), above.
- **Pages to fixed buttons:** Sometimes a designer wants to restrict a single label or multiple labels (either on the same screen or different screens) to a subset of the possible buttons (e.g., always put left engine information on the left side of the MFD screen). MFDTool allows the designer to specify as many of these constraints as desired. This constraint accommodates guideline (3) above, but also allows for more general restrictions.
- **Path movement time:** The use of some MFDs requires users to retrieve certain combinations of information. If a user has to first check the status of one system, then the status of a second, and then the status of a third, there will be a path of visited pages that correspond to this combination of information searches. Moreover, because the system information may be scattered across the MFD hierarchy, designers often include hyperlinks, or shortcuts, to the top of the hierarchy or to other MFD hierarchy locations. MFDTool allows the designer to identify these paths and acts to assign page labels to buttons to minimize the time required to execute these sequences. MFDTool allows the designer to specify as many of these paths as desired.

In MFDTool, each constraint has a corresponding numerical cost function that measures how poorly a constraint is being satisfied by the current MFD design. Larger cost values correspond to worse designs. An optimization algorithm searches through a variety of MFD designs to find one that minimizes (or nearly so) the sum of costs. The calculation of costs is described in the next section. The next section provides a mathematical description of costs.

Constraint costs

There are four types of constraints, as described in the previous section. MFDTool acts to minimize cost functions associated with these constraints. This section mathematically defines the cost functions.

Global movement time

In MFDTool, insuring that needed information can be retrieved as quickly as possible corresponds to placing MFD labels on buttons to minimize the time needed to execute the movements. For a user new to the use of a particular MFD, the savings of such minimization may be small, as much of the searching time involves reading labels and identify which buttons to press. However, for an expert user, most of the search time consists of executing the already known sequences of button presses. Identifying which button press sequences are fastest and assigning frequently searched for items to those button press sequences can lead to substantial reductions in access time.

Applying this approach requires a means of predicting how long it will take an expert user to execute a sequence of button presses. MFDs can be used with a variety of interactions (e.g., mouse clicks, finger-pointing, multiple-finger movements, special pointer pens; step cursor control, hand-on-throttle). Models for different types of interactions are dramatically different. At the moment, MFDTTool supports only finger-pointing movements because there is a well established model of how long it takes people to move a pointer over a given distance to a target of a given size. MFDTTool uses a form of Fitts' Law (Fitts, 1954; MacKenzie, 1995) that says that the movement time, M is:

$$M = I_m \log_2 \left(\frac{2D}{S} + 1 \right)$$

Here, D is the distance between the starting position of the finger and the target; S is the size of the target (MFDTTool measures this as the minimum of button height and width), \log_2 is the logarithm in base 2, and I_m is a parameter with units milliseconds/bit. I_m is empirically measured, and, for finger movements, values between 70 and 120 ms/bit are common. MFDTTool uses $I_m=100$ ms/bit.

When a sequence of movements is to be executed, MFDTTool makes the simplifying assumption that it can add up the M terms for movement from the first button to the second, the second to the third, and so on. Thus, the total time needed to execute a sequence of button presses will be

$$T = \sum_{i=1}^{m-1} M_{i,i+1},$$

where there are m button presses in the sequence, and $M_{i,i+1}$ is the time to move between successive buttons. This is almost surely a lower limit of execution time, as a user may need to read labels to remember which button to press next. There is no *a priori* way to know when a user will memorize the pattern of button presses to retrieve particular information. Such memorization surely depends on the semantics of the hierarchy and the user's experience. MFDTTool has no way to model these effects.

Future versions of MFDTTool will include support for other types of interactions, including movement with a mouse, pointer pens, step cursor control, and hand-on-throttle control. A more difficult task is to model multiple-finger movements (e.g., typing or piano playing), though it may be possible in certain situations.

Once the interaction model is defined, MFDTTool can predict how long it will take to reach a desired information label by looking at the sequence of button pushes necessary to reach that page label from the top level of the MFD hierarchy. The cost function for global movement time is the average time to reach an MFD page label:

$$C_1 = \sum_{j=1}^n T_j P_j$$

where n is the number of information labels in the hierarchy, T_j is the total time needed to execute the sequence of button presses to reach page label j , and p_j is the proportion of time that page label j is needed by the user. As the assignment of page labels to buttons is modified, the value of T_j changes. MFDTTool tries to assign page labels to button presses so that labels with larger p_j values have smaller T_j values, thereby minimizing search time.

Pages to close buttons

One could imagine a situation where a user is very knowledgeable about searching through an MFD and has memorized all the button presses to reach every page label. In such a situation, the best the designer can do is to minimize the total movement time using C_1 . However, such situations are rare. Even experienced users probably use feedback from the MFD screen to guide their searches for all but the most commonly used page labels. As a result, the designer needs to provide order among the assignment of labels to buttons that will help guide the user's search. A commonly used technique is to place labels that are related to each other on nearby buttons. A designer may, for example, want to create ordered lists of items on a single MFD screen and may also want to insure that related labels on different screens are associated with nearby buttons.

In its present version, MFDTTool defines “closeness” relative to the time needed to move between buttons. Thus, if a designer constrains page labels L_1, \dots, L_k to be as close as possible, and each label is currently assigned to buttons $b(L_1), \dots, b(L_k)$, then the quantitative cost of these assignments is:

$$C_2 = \sum_{i=1}^k \sum_{j=i+1}^k M[b(L_i), b(L_j)].$$

Here $M[b(L_i), b(L_j)]$ is the time needed to move from button $b(L_i)$ to button $b(L_j)$, as defined above. The second summation starts at $i+1$ to avoid double summation of time for each button pair. If the labels in this constraint are all on different pages, then C_2 equals zero when every label is associated with the same button. If some of the labels are on the same screen, C_2 has a nonzero minimum, as two labels cannot simultaneously be associated with the same button on the same screen. Whichever the case, MFDTTool tries to assign information labels to buttons in a way that minimizes C_2 .

Pages to fixed buttons

Sometimes a designer may want to constrain some page labels to a particular button or set of buttons. This could occur for example, if a designer, to stay consistent with other displays, wants an *EXIT* label always placed on the lower left button. Or, a designer may want geographical topics to have corresponding positions on the MFD screen (e.g., left to left). All of these constraints can be imposed in MFDTTool.

This constraint's cost measures how close the page labels are to their restricted buttons. As with the other costs, MFDTTool defines “closeness” relative to the time needed to move between buttons. Thus, if the designer constrains page labels L_1, \dots, L_k to be restricted to buttons b_1, \dots, b_h ,

and each label is currently assigned to buttons $b(L_1), \dots, b(L_k)$ then the quantitative cost of these assignments is:

$$C_3 = \sum_{i=1}^k \min_{j=1, \dots, h} M[b_j, b(L_i)].$$

The term inside the summation compares the currently assigned button for label L_i with each of the allowable buttons and takes the minimum movement time. Thus, if all labels are assigned to one of the allowable buttons, the minimum movement times are zero and the total cost is zero. When a constrained label is not assigned to an allowable button, the cost is incremented by the minimum movement time needed to move from the assigned button to one of the allowable buttons.

Path movement time

C_1 , above, measures the average time required to search for a page label, starting from the top of the hierarchy and taking the most direct route to that label. However, depending on the MFD, not all searches are of that type. It is frequently the case that a user needs to gather a number of different types of information from different screens in the MFD. The designer may include shortcuts or hyperlinks that allow the user to quickly travel along such paths of pages. Cost C_1 cannot account for these types of situations because the use of shortcuts means that there are multiple (usually infinitely many) ways to reach a label. For these types of situations, the designer must specify the sequence, or path, of pages the user goes through to perform a required task. Once this path is specified, MFDTool acts to minimize movement time along that path by associating page labels to buttons, much as for cost C_1 .

The quantitative definition of cost is much as for C_1 , except the designer must identify the path of page labels that the user steps through (for C_1 the computer could do this because each page label has a unique position in the hierarchy). The designer identifies an ordered sequence of page labels L_1, \dots, L_k for which movement time is to be minimized. If each page label is currently assigned to buttons $b(L_1), \dots, b(L_k)$, then the quantitative cost of these assignments is:

$$C_4 = \sum_{i=1}^{k-1} M[b(L_i), b(L_{i+1})].$$

MFDTool tries to minimize this cost through the assignment of page labels to buttons.

In some MFD applications, minimization of movement time along these paths may be the most important job for the designer. By their very nature, such sequences must be specified by the designer.

Weighting costs

All of the constraint costs are defined in terms of milliseconds of time needed to move between buttons. However, the designer still needs to identify the relative importance of different constraints so that MFDTool produces the desired result. It is common for constraints to be in

conflict with each other. In anticipation of such conflicts the designer needs to indicate a weight, λ , for each constraint cost. For example, if the designer wants to be certain that the *EXIT* label is always on the lower left button, even if such assignment means an increase in average search time, then the weight for the *EXIT* constraint might be set larger than the weight for the average search time.

MFDTool tries to minimize the weighted sum of constraint costs:

$$C = \sum_{i=1}^n \lambda_i R_i.$$

Here, there are n constraints defined by the user, and R_i corresponds to the cost associated with constraint i .

There is no way for MFDTool to advise the designer on how to set the weights. The default is the value one, but it is merely a starting point and not intended as a reasonable choice. The values of the weights have a great effect on the resulting MFD design, and it is not unusual for a designer to tweak the weights to insure that one constraint is satisfied over another. The use of extremely large weights, relative to others, is often not effective because it sometimes hinders the optimization process (next section).

Optimization

Once a total cost function is defined, one can use any number of algorithms to find the assignment of page labels to buttons that minimizes that cost function. MFDTool currently uses the simulated annealing algorithm, but future versions of MFDTool may explore other approaches.

Simulated annealing is a variation of hill-climbing algorithms. In a hill-climbing (or hill-descending, only the sign needs to be changed) algorithm, the system is initialized to a particular state (e.g., mapping of labels to buttons) and the cost is calculated for that state. One of the variables of the problem (e.g., a label) is randomly selected and modified (e.g., moved to a new button). A new cost value is calculated, and if the new cost is less than the old cost, the change is kept, otherwise the change is undone. In this way, the system converges to a state where any change would lead to an increase in cost (e.g., where any change in the mapping would be worse). Hill-climbing techniques have a tendency to get stuck in local minima of cost because they never accept changes that increase cost. In complex problems, hill-climbing methods can easily get trapped in a state where any change only increases cost but the global minimum is very different, with a much smaller cost. What is needed is a controlled way to climb out of local minima and end in a state with the global minimum of cost. By analogy, one would probably, at some point during a hike, need to go down a ravine or a small slope to climb to the top of a mountain.

Simulated annealing is a stochastic algorithm that at first accepts changes even if they lead to larger costs. As time progresses, a temperature parameter gradually decreases (this is the annealing) so that it becomes less likely that a change leading to an increase in cost is accepted.

As the temperature becomes small the algorithm becomes essentially hill-climbing. As long as the temperature decreases slowly enough and enough changes are considered at each temperature level, simulated annealing is statistically guaranteed to find the global minimum of a problem. In practice, though, the necessary temperature schedule is too slow and the number of changes at each level is too big, so simpler approaches are taken that are faster, but less certain to find the global minimum.

In simulated annealing, the initial temperature, T , is set large enough that many state changes are accepted even if they lead to a cost increase. MFDTool sets the initial temperature in the following way. Given the state of the system at the start of the optimization process, many (50 times the number of page labels) changes are made to the MFD, and the change in cost is calculated for each change. The average of these cost changes is the initial temperature for the annealing process. The final state of the system after all these changes is also the initial state for the start of the annealing process.

Changes are made by randomly selecting an MFD page label. Its button assignment is noted, and a new button assignment is randomly selected. The selected label swaps positions with whatever (perhaps nothing) is at the new button assignment. After each change, new cost, C_{new} , is calculated and compared to the cost before the change, C_{old} . The change in cost, $\Delta C = C_{new} - C_{old}$, is calculated. If the cost change is negative, the change is kept. If the cost change is positive, the change is kept when a random number between zero and one is greater than

$$p = \exp(-\Delta C / T)$$

This relationship means that when ΔC is much smaller than T , p is close to one, and lots of changes are kept. As T gets smaller than ΔC , p gets closer to zero, and changes are not kept very often. Statistically then, the system is more likely to be in a state with a low cost. As T decreases, the system tends to be stuck in a state with very low cost.

To insure that the statistical situation is close to reality, one needs to implement many changes at every temperature level. MFDTool makes 300 times the number of page labels changes at every temperature level. After these changes, the temperature is modified by the equation

$$T_{new} = 0.99T_{old}$$

The process is then repeated for the new temperature. The whole process stops when it seems that the temperature is so small that the system is trapped in a particular state (as in hill-climbing). MFDTool reaches this conclusion when 10 changes in temperature have not produced any changes in cost.

At the end of the simulated annealing process, the system should be in a state with a low (but perhaps not optimal) cost. Being certain of finding the true optimal state with the absolute lowest possible cost would be prohibitively difficult and would likely require a supercomputer, even for relatively small MFDs.

MFDTTool is written in the Java programming language. Source code is provided in Appendix A of this document. A full description of MFDTTool, and the procedures to use it, is provided in the MFDTTool User's Guide, which is attached as Appendix B of this document.

Conclusions

Previous work showed that the computational approach described in Francis and Reardon (1977) and further developed in Francis (1998) had operational benefit. However, the programming skills needed to implement the algorithms from scratch are difficult to come by. MFDTTool provides a means whereby experts in MFD design, but not necessarily experts in optimization and computer programming, can use the computational algorithms to guide their design process. We anticipate that MFDTTool will prove valuable to many MFD designers in a variety of different contexts.

References

- Calhoun, G. 1978. Control logic design criteria for multifunction switching devices. In: Proceedings of the Human Factors Society 22nd Annual Meeting: 383-387.
- Cook, R. and Woods, D. 1996. Adapting to new technology in the operating room. Human Factors. 38: 593-613.
- Cuomo, D. L., Borghesani, L., Khan, K. and Violett, D. 1998. Navigating the company web. Ergonomics in Design, 6, 7-14.
- Department of Defense. 1981. Military standard: Human engineering design criteria for military systems, equipment, and facilities. MIL-STD-1472D.
- Dohme, J. 1995. The military quest for flight training effectiveness. Vertical Flight Training. W. Larsen, R. Randle, and L. Popish (Eds.) NASA Reference Publication 1373.
- Farrell, P. 1997. A human-machine interaction analysis using layered protocol theory. Proceedings of the 29th Annual Conference of HFAC/ACE, Winnipeg, Manitoba, 39-41.
- Fisher, D. L. 1993. Optimal performance engineering: Good, better, best. Human Factors, 35, 115-139.
- Fisher, D., Yungkurth, E., and Moss, S. 1990. Optimal menu hierarchy design: Syntax and semantics. Human Factors. 32: 665-683.
- Fitts, P. M. 1954. The information capacity of the human motor system in controlling the amplitude of movement. Journal of Experimental Psychology. 47: 381-391.
- Francis, G. 1998. Designing optimal hierarchies for information retrieval with multifunction displays Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 98-33.
- Francis, G. and Reardon, M. 1997. Aircraft multifunction display and control systems: A new quantitative human factors design method for organizing functions and display contents Fort Rucker, AL: U.S. Army Aeromedical Research Laboratory. USAARL Report No. 97-18.
- Holley, C. and Busbridge, M. 1995. Evolution of the Venom variant of the AH-1W Supercockpit. Proceedings of the American Helicopter Society 51st Annual Forum. 1436-1449.
- Hwa, R., Marks, J. and Shieber, S. 1995. Automatic structuring of embedded hypermedia documents. TR-95-6, Mitsubishi Electric Research Laboratory.

- Lind, J. 1981. Evaluation of cockpit procedures, displays, and controls for stores management in the advanced aircraft armament systems (AAAS).. Naval Weapons Center Technical Memorandum 4538.
- MacKenzie, I. S. 1995. Movement time prediction in human-computer interfaces. In Readings in human-computer interactions (2nd ed.), R. M. Baecker, W. A. S. Buxton, J. Grudin, & S. Greenberg (Eds.). Kaufmann: Los Altos, CA.
- Obradovich, J. H. and Woods, D. D. 1996. Users as designers: How people cope with poor HCI design in computer-based medical devices. Human Factors. 38: 574-592.
- Reising, J., and Curry, D. 1987. A comparison of voice and multifunction controls: Logic design is the key. Ergonomics. 30: 1063-1077.
- Rogers, W., Cabrera, E., Walker, N., Gilbert, K., and Fisk, A. 1996. A survey of automatic teller machine usage across the adult life span. Human Factors. 38: 156-166.
- Roske-Hofstrand, R. and Paap, K. 1986. Cognitive networks as a guide to menu organization. Ergonomics. 29 1301-1311.
- Sargent, T. A., Kay, M. G., and Sargent, R. G. 1997. A methodology for optimally designing console panels for use by a single operator. Human Factors, 39, 389-409.
- Sanders, M. and McCormick, E. 1987. Human Factors in Engineering and Design. New York, NY: McGraw-Hill Book Co.
- Sirevaag, E., Kramer, A., Wickens, C., Reisweber, M., Strayer, D., and Grenell, J. 1993. Assessment of pilot performance and mental workload in rotary wing aircraft. Ergonomics. 36: 1121-1140.
- Snowberry, K., Parkinson, S. R., and Sisson, N. 1983. Computer display menus. Ergonomics 26(7), 699-712.
- Spiger, R. and Farrell, R. 1982. Survey of multifunction display and control technology. NASA-CR-167510.
- Tietlebaum, R. and Granda, R. 1983. The effects of positional constancy on searching menus for information. In: Proceedings of Human Factors in Computing Systems. New York, NY: Association for Computing Machinery.
- Williges, R., Williges, B., and Fainter, R. 1988. Software interfaces for aviation systems. In Human Factors in Aviation, E. Wiener and D. Nagel (Eds.). Academic Press: San Diego.

Appendix A.

Java source code.

This appendix provides the source code of each Java class.

CloseableFrame

```
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;
```

```
/* class CloseableFrame.class
```

This object provides a window that, when the close window command is given, hides the window.

Written by Greg Francis, Purdue University
August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
```

```
public class CloseableFrame extends Frame implements WindowListener, Serializable
```

```
{
    public CloseableFrame(String title)
    {
        setSize( 200, 200 );
        setTitle( title );
        addWindowListener( this );
    }

    // close the window when requested
    public void windowClosing( WindowEvent e )
    {
        this.dispose();
    }

    // the remaining methods must be defined, but do not do anything
    public void windowClosed( WindowEvent e )
    { }
    public void windowIconified( WindowEvent e )
    { }
    public void windowOpened( WindowEvent e )
    { }
    public void windowDeiconified( WindowEvent e )
    { }
    public void windowActivated( WindowEvent e )
    { }
    public void windowDeactivated( WindowEvent e )
    { }

    public static void main( String args[] )
    { Frame f = new CloseableFrame("Closeable frame");
      f.show();
    }
}
```

MFDPage

```
/* class MFDPage.class
```

This object holds information about an MFD page. Written by Greg Francis, Purdue University, August 1999 for US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
import java.util.Vector;
import java.io.Serializable;

public class MFDPPage implements Serializable
{
    public static final int TERMINATOR=0, PARENT=1, HYPERLINK=2; // type of page
    String Name;
    Vector Siblings= new Vector();
    MFDPPage Parent; // has to be assigned after initial creation
    int TypeOfPage;
    MFDPPage HyperLink = null;
    int ButtonAssignment;
    double Proportion;
    boolean fixedProportion;

    // Parent
    public MFDPPage(String Name,int typeOfPage, Vector Siblings)
    {
        this.Name = Name;
        this.Parent = Parent;
        this.TypeOfPage = typeOfPage;
        if(TypeOfPage == PARENT)
            this.Siblings = Siblings;
    }

    // Terminator
    public MFDPPage(String Name, int typeOfPage)
    {
        this.Name = Name;
        this.Parent = Parent;
        this.TypeOfPage = typeOfPage;
    }

    // Hyperlink
    public MFDPPage(String Name, int typeOfPage, MFDPPage hyperLink)
    {
        this.Name = Name;
        this.Parent = Parent;
        this.TypeOfPage = typeOfPage;
        if(TypeOfPage == HYPERLINK)
            this.HyperLink = hyperLink;
    }
}
```

MFDButton

```
/* class MFDButton.class
```

This object provides a button with a designer-specified size and location. All physical measurements are in inches.

Written by Greg Francis, Purdue University
August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```
*/
import java.awt.Button;
import java.io.Serializable;
```

```

public class MFDButton extends Button implements Serializable
{
    Button button;
    float xSize, ySize; // sizes are in inches
    float xPosition, yPosition;

    public MFDButton(String Name, float xSize, float ySize, float xPosition, float yPosition)
    {
        this.button = new Button(Name);
        this.xSize = xSize;
        this.ySize = ySize;
        this.xPosition = xPosition;
        this.yPosition = yPosition;
    }
}

```

MFDOptimize

```

/* class MFDOptimize.class

```

This object applies the optimization algorithms to a provided MFD system.

Written by Greg Francis, Purdue University
 August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author
 and should not be construed as an official Department of the Army position, or decision,
 unless so designated by other documentation.

```

*/

```

```

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.Random;
import java.io.Serializable;

public class MFDOptimize implements Serializable
{
    private String lastDir = "";

    MFD mfd;
    Vector Pages;
    MFDPage currentPage;
    int Movement[][]; // pairwise movement time

    public MFDOptimize(MFD mfd)
    {
        this.mfd = mfd;
        this.Pages = mfd.mfdhierarchy.Pages;

        Movement = new int [mfd.mfdframe.Buttons.size()][mfd.mfdframe.Buttons.size()];
        ComputeTimeToMove(mfd.mfdframe.Buttons, Movement);
        mfd.upDate.setText("Done computing movement times.");

        // Do optimization
        Optimize(Pages, Movement, mfd.mfdframe.Buttons);
    }

    public void keyTyped( KeyEvent evt ) { }
    public void keyReleased( KeyEvent evt ) { }
    public void keyPressed( KeyEvent evt ) { }

    public void ComputeTimeToMove(Vector buttons, int [][] Movement)
    {
        int bsize = buttons.size();

```

```

// Find all movement times with Fitts' law
for(int i=0;i<bSize;i++)
{
    MFDButton firstB = (MFDButton)buttons.elementAt(i);
    // middle of first button
    double x1 = firstB.xPosition + firstB.xSize/2.0;
    double y1 = firstB.yPosition + firstB.ySize/2.0;
    for(int j=0;j<bSize;j++)
    {
        MFDButton secondB = (MFDButton)buttons.elementAt(j);
        // middle of second button
        double x2 = secondB.xPosition + secondB.xSize/2.0;
        double y2 = secondB.yPosition + secondB.ySize/2.0;

        // calculate Euclidean distance
        double distance = Math.sqrt( Math.pow(x1-x2,2.0) + Math.pow(y1-y2,2.0) );

        // Calculate movement time with b=100 ms/bits for finger movement (Mackenzie et al, 1991)
        Movement[i][j] = (int)(100 * Math.log(2*distance/Math.min(secondB.xSize, secondB.ySize) + 1.0)/Math.log(2.0) );
    }
}
}

public void Optimize(Vector Pages, int [][] Movement, Vector buttons)
{
    Random randGen = new Random();
    double CurrentCost = Cost(mfd.Constraints, Movement);
    int bSize = buttons.size();
    boolean doOptimize=true;
    int numTempChanges = 0;
    float Temperature = (float)0.0;

    mfd.upDate.setText("Initalizing...");
    // initialize to a random state and find initial temperature as average of changes
    int pageSize = Pages.size();
    for(int swapNum=0;swapNum<50*pageSize;swapNum++)
    {
        // pick a random Page that is a parent
        int tempIndex = (int)Math.abs(randGen.nextInt())%(pageSize);
        MFDPPage parent = (MFDPPage)Pages.elementAt(tempIndex);
        while(!(parent.TypeOfPage==MFDPPage.PARENT) || parent.Siblings.size()==0)
        {
            tempIndex = (int)Math.abs(randGen.nextInt())%(pageSize);
            parent = (MFDPPage)Pages.elementAt(tempIndex);
        }
        int SibSize = parent.Siblings.size();

        // Randomly pick a sibling and a new location
        int sibIndex1 = (int)Math.abs(randGen.nextInt())%(SibSize);
        MFDPPage sib1 = (MFDPPage)parent.Siblings.elementAt(sibIndex1);

        int newPosition = (int)Math.abs(randGen.nextInt())%(bSize);
        while(sib1.ButtonAssignment==newPosition)
            newPosition = (int)Math.abs(randGen.nextInt())%(bSize);

        // check whether there is a page at position to swap into
        boolean swapToEmpty=true;
        MFDPPage sib2=null;
        for(int i=0;i<SibSize;i++)
        {
            sib2= (MFDPPage)parent.Siblings.elementAt(i);
            if(sib2.ButtonAssignment==newPosition)
            {
                swapToEmpty=false;
                i=SibSize;
            }
        }

        int swapFrom= -1;
        if(swapToEmpty)

```

```

    {
        swapFrom = sib1.ButtonAssignment;
        sib1.ButtonAssignment = newPosition;
    }
    else
    {
        int temporary = sib1.ButtonAssignment;
        sib1.ButtonAssignment = sib2.ButtonAssignment;
        sib2.ButtonAssignment = temporary;
    }

    // check new cost, keep swap regardless (randomizing state)
    double NewCost = Cost(mfd.Constraints, Movement);
    double ChangeCost = NewCost - CurrentCost;

    // for computing initial temperature later
    if(NewCost > CurrentCost)
    {
        Temperature += ChangeCost;
        numTempChanges++;
    }
}

// set initial temperature
Temperature /= numTempChanges;

int stepsSinceChangedCost=0; // to note when system seems to have converged

while(doOptimize)
{
    double startCost = CurrentCost;
    Temperature *= 0.99;
    for(int swapNum=0; swapNum<300*pageSize; swapNum++)
    {
        // pick a random Page that is a parent
        int tempIndex = (int)Math.abs(randGen.nextInt())%(pageSize);
        MFDPage parent = (MFDPage)Pages.elementAt(tempIndex);
        while(!(parent.TypeOfPage==MFDPage.PARENT) || parent.Siblings.size()==0)
        {
            tempIndex = (int)Math.abs(randGen.nextInt())%(pageSize);
            parent = (MFDPage)Pages.elementAt(tempIndex);
        }
        int SibSize = parent.Siblings.size();

        // Randomly pick a sibling and a new location
        int sibIndex1 = (int)Math.abs(randGen.nextInt())%(SibSize);
        MFDPage sib1 = (MFDPage)parent.Siblings.elementAt(sibIndex1);

        int newPosition = (int)Math.abs(randGen.nextInt())%(bSize);
        while(sib1.ButtonAssignment==newPosition)
            newPosition = (int)Math.abs(randGen.nextInt())%(bSize);

        // check whether there is a page at position to swap into
        boolean swapToEmpty=true;
        MFDPage sib2=null;
        for(int i=0; i<SibSize; i++)
        {
            sib2= (MFDPage)parent.Siblings.elementAt(i);
            if(sib2.ButtonAssignment==newPosition)
            {
                swapToEmpty=false;
                i=SibSize;
            }
        }

        int swapFrom=-1;
        if(swapToEmpty)
        {
            swapFrom = sib1.ButtonAssignment;
            sib1.ButtonAssignment = newPosition;

```

```

    }
    else
    {
        int temporary = sib1.ButtonAssignment;
        sib1.ButtonAssignment = sib2.ButtonAssignment;
        sib2.ButtonAssignment = temporary;
    }

    // check new cost and decide whether to keep change
    double NewCost = Cost(mfd.Constraints, Movement);
    double ChangeCost = Math.max(NewCost - CurrentCost, 0.0);
    double probSwaptemp = Math.exp(-ChangeCost/(double)Temperature);
    double prob = randGen.nextDouble();

    if(ChangeCost>0 && probSwaptemp < prob) // if true, swap back
    {
        if(swapToEmpty)
            sib1.ButtonAssignment = swapFrom;
        else
        {
            int temporary = sib1.ButtonAssignment;
            sib1.ButtonAssignment = sib2.ButtonAssignment;
            sib2.ButtonAssignment = temporary;
        }
    }
    else // keep swap
        CurrentCost = NewCost;
}

if(startCost==CurrentCost) // no change after full run at this temperature
    stepsSinceChangedCost++;
else
    startCost = CurrentCost;
mfd.upDate.setText("Cost="+CurrentCost+". Temp="+Temperature+" Cost unchanged for "+stepsSinceChangedCost+"
temperature levels");
if(stepsSinceChangedCost==10) // stop search, system has probably converged
    doOptimize=false;
}
mfd.upDate.setText("Final cost = "+CurrentCost);
}

public double Cost(Vector Constraints, int [][] Movement)
{
    double cost=0.0;
    int constraintSize= Constraints.size();
    for(int i=0;i<constraintSize;i++)
    {
        MFDConstraint currentConstraint = (MFDConstraint)Constraints.elementAt(i);
        cost += currentConstraint.Cost(Movement);
    }
    return(cost);
}

public static void main(String[] args)
{
    MFD f = new MFD("MFD");
    f.setVisible(true);
}
}

```

MFDConstraint

/* class MFDConstraint.class

This object contains information and methods for dealing with MFD constraints.

Written by Greg Francis, Purdue University
August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

*/

```
import java.util.Vector;
import java.io.Serializable;

public class MFDCConstraint implements Serializable
{
    public static final int MOVEMENTTIME=0, PATHMOVEMENTTIME=1, RELATEDNEARBY=2, RESTRICTEDPLACES=3;
    String Name;
    Vector Pages= new Vector();
    int [] Places = new int[1];
    int TypeOfConstraint;
    double Weight;

    // Not restricted places
    public MFDCConstraint(String Name, Vector applicablePages, int TypeOfConstraint, double Weight)
    {
        this.Name = Name;
        this.TypeOfConstraint = TypeOfConstraint;
        this.Pages = applicablePages;
        this.Weight = Weight;
    }

    // Restricted places requires Vector of places
    public MFDCConstraint(String Name, Vector applicablePages, int TypeOfConstraint, double Weight, int [] Places)
    {
        this.Name = Name;
        this.TypeOfConstraint = TypeOfConstraint;
        this.Pages = applicablePages;
        this.Weight = Weight;
        if(TypeOfConstraint==RESTRICTEDPLACES)
            this.Places = Places;
    }

    // Method for calculating constraint cost
    public double Cost(int [][] Movement)
    {
        double cost = 0.0;
        int pageSize = Pages.size();

        switch(TypeOfConstraint)
        {
            case MOVEMENTTIME:
                for(int i=0;i<pageSize;i++)
                {
                    MFDPPage currentPage = (MFDPPage)Pages.elementAt(i);
                    int TimeForPath=0;

                    // go backwards through path
                    while(!currentPage.equals(currentPage.Parent))
                    {
                        TimeForPath += Movement[currentPage.Parent.ButtonAssignment][currentPage.ButtonAssignment];

                        currentPage = currentPage.Parent;
                    }
                    currentPage = (MFDPPage)Pages.elementAt(i);
                    cost += currentPage.Proportion * TimeForPath;
                }
                break;

            case PATHMOVEMENTTIME:
                for(int i=0;i<pageSize-1;i++)
                {
                    MFDPPage page1 = (MFDPPage)Pages.elementAt(i);
                    MFDPPage page2 = (MFDPPage)Pages.elementAt(i+1);
```

```

        cost += Movement[page1.ButtonAssignment][page2.ButtonAssignment];
    }
    break;

    case RELATEDNEARBY:
    for(int i=0;i<pageSize;i++)
    {
        MFDPage page1 = (MFDPage)Pages.elementAt(i);
        for(int j=i+1;j<pageSize;j++)
        {
            MFDPage page2 = (MFDPage)Pages.elementAt(j);
            cost += Movement[page1.ButtonAssignment][page2.ButtonAssignment];
        }
    }
    break;

    case RESTRICTEDPLACES:
    for(int i=0;i<pageSize;i++)
    {
        MFDPage page1 = (MFDPage)Pages.elementAt(i);
        int buttonPlaces = Places.length;
        float pageCost = Float.POSITIVE_INFINITY;
        for(int j=0;j<buttonPlaces;j++)
        {
            if(pageCost>Movement[page1.ButtonAssignment][Places[j]])
            {
                pageCost = (float)Movement[page1.ButtonAssignment][Places[j]];
            }
        }
        cost+=pageCost;
    }
    break;
}
return(Weight*cost);
}
}
}

```

MFDFrame

/* class MFDFrame.class

This object provides a window for the emulation of the physical MFD and its buttons.

Written by Greg Francis, Purdue University
August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

*/

```

import java.awt.*;
import java.util.Vector;
import java.awt.event.*;
import java.io.Serializable;

```

```

public class MFDFrame extends CloseableFrame implements ActionListener, Serializable
{
    String Name, Interaction;
    float xSize, ySize;
    Vector Buttons;
    MFD mfd;

    public MFDFrame(MFD mfd, String Name, String Interaction, float xSize, float ySize, Vector Buttons)
    {
        super("MFD Hardware: "+Name);
    }
}

```

```

this.mfd = mfd;
this.Name = Name;
this.Interaction = Interaction;
this.xSize = xSize;
this.ySize = ySize;
this.Buttons = Buttons;

Dimension screenSize =Toolkit.getDefaultToolkit().getScreenSize();
float screenResolution = Toolkit.getDefaultToolkit().getScreenResolution();

this.setLayout(new GridLayout(1,1));
Panel panel = new Panel();
panel.setLayout(null);
this.add(panel);

this.setSize((int)(screenResolution*xSize)+getInsets().left+getInsets().right,(int)(screenResolution*ySize)+getInsets().top+getInsets().bottom
);

this.setResizable(false);
// Add buttons to frame
for(int i=0;i<Buttons.size();i++)
{
    panel.add(((MFDButton)Buttons.elementAt(i)).button);

    ((MFDButton)Buttons.elementAt(i)).button.setBounds((int)(screenResolution*((MFDButton)Buttons.elementAt(i)).xPosition),
(int)(screenResolution*((MFDButton)Buttons.elementAt(i)).yPosition), (int)(screenResolution*((MFDButton)Buttons.elementAt(i)).xSize),
(int)(screenResolution*((MFDButton)Buttons.elementAt(i)).ySize));

    ((MFDButton)Buttons.elementAt(i)).button.addActionListener(this);
}
this.show();
}

public static void main(String[] args)
{
    MFD f = new MFD("MFD");
    f.setVisible(true);
}

public void ShowActivePage(MFDPage currentPage)
{
    // clear all the buttons
    for(int i=0;i<Buttons.size();i++)
        ((MFDButton)Buttons.elementAt(i)).button.setLabel(" ");

    if(currentPage.TypeOfPage == MFDPage.PARENT)
    {
        for(int i=0;i<currentPage.Siblings.size();i++)
        {
            MFDPage tempSib = (MFDPage)currentPage.Siblings.elementAt(i);
            if(tempSib.TypeOfPage == MFDPage.PARENT)
                ((MFDButton)Buttons.elementAt(tempSib.ButtonAssignment)).button.setLabel(tempSib.Name+" >");
            else
                ((MFDButton)Buttons.elementAt(tempSib.ButtonAssignment)).button.setLabel(tempSib.Name);
        }
    }
    else if(currentPage.TypeOfPage == MFDPage.TERMINATOR)
    {
        // show parent
        ShowActivePage (currentPage.Parent);
    }
    else if(currentPage.TypeOfPage == MFDPage.HYPERLINK)
    {
        if(currentPage.HyperLink==null) // if hyperlink not defined
            ShowActivePage(currentPage.Parent);
        else // show hyperlink
            ShowActivePage (currentPage.HyperLink);
    }
    repaint();
    setVisible(true);
}

```



```

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.io.Serializable;

public class MFDHierarchy extends CloseableFrame
    implements ItemListener, ActionListener, Serializable
{
    private String lastDir = "";
    MFD mfd;
    Vector Pages;
    MFDPage currentPage;
    int activePage;
    List PageList = new List();
    Button showParent;
    Label PathSummary;

    public MFDHierarchy(MFD mfd, Vector Pages)
    {
        super("Hierarchy: "+mfd.Name);
        this.mfd = mfd;
        this.Pages = Pages;

        setLayout( new BorderLayout() );
        Panel HierarchyPanel = new Panel();
        this.add(HierarchyPanel);

        setBackground( Color.white );
        this.setSize(300,300);

        PageList = new List();
        HierarchyPanel.setLayout(new BorderLayout(5,5));
        HierarchyPanel.add("Center",PageList);
        PageList.addItemListener(this);

        PathSummary = new Label(" ");
        HierarchyPanel.add("South",PathSummary);

        showParent = new Button("Show parent of page");
        HierarchyPanel.add("North",showParent);
        showParent.addActionListener(this);

        // top page
        currentPage= (MFDPage)Pages.elementAt(0);
        for(int i=1;i<Pages.size();i++)
        {
            if(currentPage.equals(currentPage.Parent))
                i=Pages.size();
            else
                currentPage = (MFDPage)Pages.elementAt(i);
        }

        ShowActivePage(currentPage);
    }

    public void ShowActivePage(MFDPage currentPage)
    {
        // put currentPage labels on far right column
        PageList.clear();
        if(currentPage.TypeOfPage == MFDPage.PARENT)
        {
            for(int i=0;i<currentPage.Siblings.size();i++)
            {
                MFDPage tempSib = (MFDPage)currentPage.Siblings.elementAt(i);
                if(tempSib.TypeOfPage == MFDPage.PARENT)
                    PageList.addItem(tempSib.Name+" >");
                else
            }
        }
    }
}

```

```

        PageList.addItem(tempSib.Name);
    }
}
else if(currentPage.TypeOfPage == MFDPage.TERMINATOR)
{
    // show parent
    ShowActivePage (currentPage.Parent);
    // set currentPage as selected
    PageList.select(currentPage.Parent.Siblings.indexOf(currentPage));
}
else if(currentPage.TypeOfPage == MFDPage.HYPERLINK)
{
    // show parent
    ShowActivePage (currentPage.Parent);
    // set currentPage as selected
    PageList.select(currentPage.Parent.Siblings.indexOf(currentPage));
    // set currentPage as selected
    PageList.select(currentPage.Parent.Siblings.indexOf(currentPage));
}
// define path
MFDPage tempPage = currentPage;
String pathname = "/" + currentPage.Name;
while(!tempPage.equals(tempPage.Parent))
{
    tempPage = tempPage.Parent;
    pathname = "/" + tempPage.Name + pathname;
}

PathSummary.setText(pathname);
repaint();
mfd.mfdframe.ShowActivePage(currentPage);
setVisible(true);
}

public static void main(String[] args)
{
    MFD f = new MFD("MFD");
    f.setVisible(true);
}

public void itemStateChanged(ItemEvent event)
{
    int selected = PageList.getSelectedIndex();
    if(selected != -1)
    {
        if(currentPage.TypeOfPage == MFDPage.PARENT)
        {
            MFDPage sibPage = (MFDPage)currentPage.Siblings.elementAt(selected);
            currentPage = sibPage;
        }
        else if(currentPage.TypeOfPage == MFDPage.TERMINATOR)
        {
            MFDPage sibPage = (MFDPage)currentPage.Parent.Siblings.elementAt(selected);
            currentPage = sibPage;
        }
        else if (currentPage.TypeOfPage == MFDPage.HYPERLINK)
        {
            MFDPage sibPage = (MFDPage)currentPage.Parent.Siblings.elementAt(selected);
            currentPage = sibPage;
            if(!(currentPage.HyperLink == null)) // if hyperlink is defined
            {
                if(currentPage.HyperLink.TypeOfPage == MFDPage.TERMINATOR)
                    currentPage = currentPage.HyperLink.Parent;
                else
                    currentPage = currentPage.HyperLink;
            }
        }
        ShowActivePage(currentPage);
    }
}

```

```

        // update list on MFD
        mfd.PageList.select(Pages.indexOf(currentPage));
        mfd.LoadPageProperties(currentPage);
    }
}

public void actionPerformed (ActionEvent event)
{
    String arg = event.getActionCommand();
    if (arg.equals("Show parent of page"))
    {
        if(currentPage.TypeOfPage==MFDPage.PARENT)
            currentPage = currentPage.Parent;
        else if(currentPage.TypeOfPage==MFDPage.TERMINATOR)
            currentPage = currentPage.Parent.Parent;
        else if(currentPage.TypeOfPage==MFDPage.HYPERLINK)
            currentPage = currentPage.Parent.Parent;
        ShowActivePage(currentPage);
        // update list on MFD
        mfd.PageList.select(Pages.indexOf(currentPage));
    }
}
}

```

MFD

/* class MFD.class

This object provides a window and user interface to open, save, and define new MFDs. It also provides an interface for defining constraints and starting the optimization.

Written by Greg Francis, Purdue University
August 1999

For US Army Aeromedical Research Laboratory, Ft. Rucker, Alabama

The views opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, or decision, unless so designated by other documentation.

```

*/

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;
import java.util.StringTokenizer;
import java.io.Serializable;

public class MFD extends CloseableFrame
    implements ActionListener, ItemListener, Serializable
{
    private String lastDir = "";
    MFDFrame mfdframe;
    MFDHierarchy mfdhierarchy;
    Vector Constraints = new Vector();
    List PageList;
    Label upDate;
    String Name;
    TextField changeProportion;
    Checkbox [] checkFixed= new Checkbox[2]; // fixed proportion or not
    TextField weight, constName;
    List showConstraints, showButtons, constrainedPagesList;
    Choice constraintChoice;
    Button saveConstraint, newConstraint, addPageConst, deletePageConst, editConstraint, deleteConstraint;
    Vector ConstrainedPages = new Vector();
    MFDConstraint currentConstraint;
    Button setHyperlink;
    Label editLabel, nameHyperlink;
    boolean LookingForHyperlink=false;

```

MFDPage editingPage;

```
public MFD(String title)
{
    super(title);
    setLayout( new BorderLayout() );
    MenuBar mbar = new MenuBar();
    Menu mmfd = new Menu("File");
    MenuItem m1mfd = new MenuItem("Open");
    m1mfd.addActionListener(this);
    mmfd.add(m1mfd);
    MenuItem m2mfd = new MenuItem("Save");
    m2mfd.addActionListener(this);
    mmfd.add(m2mfd);
    mbar.add(mmfd);
    MenuItem m3mfd = new MenuItem("New");
    m3mfd.addActionListener(this);
    mmfd.add(m3mfd);
    mbar.add(mmfd);

    setMenuBar(mbar);

    this.setLayout(new BorderLayout(5,5));
    PageList = new List();
    PageList.addItemListener(this);
    this.add("West",PageList);

    Panel top = new Panel();
    top.setLayout(new BorderLayout(5,5));

    Panel editProportion = new Panel();
    editProportion.setLayout(new GridLayout(2,1,2,2));
    editLabel = new Label("Edit proportion of time user targets this page: ");
    editProportion.add(editLabel);
    changeProportion = new TextField(" ");
    editProportion.add(changeProportion);

    Panel hyperlink = new Panel();
    hyperlink.setLayout(new GridLayout(1,2,2,2));
    setHyperlink = new Button("Change hyperlink");
    nameHyperlink = new Label(" ");
    hyperlink.add(setHyperlink);
    hyperlink.add(nameHyperlink);
    setHyperlink.addActionListener(this);
    setHyperlink.disable();

    top.add("North", editProportion);
    top.add("Center", hyperlink);
    Button saveChanges = new Button("Save changes to page");
    top.add("South", saveChanges);
    saveChanges.addActionListener(this);

    CheckboxGroup CbG= new CheckboxGroup();
    checkFixed[1] = new Checkbox("Fixed", CbG, false);
    checkFixed[0] = new Checkbox("Free", CbG, true);

    Panel cbgPanel = new Panel();
    cbgPanel.setLayout(new GridLayout(3,1));
    cbgPanel.add(new Label("Restriction on proportion"));
    cbgPanel.add(checkFixed[0]);
    cbgPanel.add(checkFixed[1]);

    top.add("East", cbgPanel);
    this.add("North",top);

    // Set up layout for constraints
    Panel Optimize = new Panel();
    Optimize.setLayout(new BorderLayout(5,5));

    Panel ListConst = new Panel();
```

```

ListConst.setLayout(new BorderLayout());
showConstraints = new List();
ListConst.add("North", new Label("Select constraint"));
ListConst.add("Center",showConstraints);
editConstraint = new Button("Edit constraint");
deleteConstraint = new Button("Delete constraint");
Panel editDeleteC = new Panel();
editDeleteC.setLayout(new GridLayout(1,2));
editDeleteC.add(deleteConstraint);
editDeleteC.add(editConstraint);
ListConst.add("South", editDeleteC);
editConstraint.addActionListener(this);
deleteConstraint.addActionListener(this);

Panel ButtonsPanel = new Panel();
ButtonsPanel.setLayout(new BorderLayout());
ButtonsPanel.add("North",new Label("Select buttons"));
showButtons = new List();
showButtons.setMultipleSelections(true);
ButtonsPanel.add("Center",showButtons);

Panel leftConst = new Panel();
leftConst.setLayout(new GridLayout(2,1,5,10));
leftConst.add(ListConst);
leftConst.add(ButtonsPanel);

Optimize.add("West",leftConst);

Panel constP = new Panel();
constP.setLayout(new BorderLayout());
constP.add("North",new Label("Add a new constraint"));
constraintChoice = new Choice();
constraintChoice.addItem("Global movement time");
constraintChoice.addItem("Path movement time");
constraintChoice.addItem("Pages to close buttons");
constraintChoice.addItem("Pages to fixed buttons");
constP.add("Center",constraintChoice);
Panel newSave = new Panel();
newSave.setLayout(new GridLayout(2,1,0,2));
newConstraint = new Button("New constraint");
saveConstraint = new Button("Save constraint");
newSave.add(newConstraint);
newSave.add(saveConstraint);
constP.add("South",newSave);
newConstraint.addActionListener(this);
saveConstraint.addActionListener(this);
newConstraint.disable();
saveConstraint.disable();

Panel pageP = new Panel();
pageP.setLayout(new BorderLayout());
pageP.add("North", new Label("Constrained pages"));
constrainedPagesList = new List();
pageP.add("Center", constrainedPagesList);
addPageConst = new Button("Add MFD page");
deletePageConst = new Button("Remove selected page");
Panel pagePanel = new Panel();
pagePanel.setLayout(new GridLayout(1,2));
pagePanel.add(deletePageConst);
pagePanel.add(addPageConst);
pageP.add("South", pagePanel);
addPageConst.addActionListener(this);
deletePageConst.addActionListener(this);
addPageConst.disable();
deletePageConst.disable();

Panel constT = new Panel();
constT.setLayout(new GridLayout(2,2));
constT.add(new Label("Weight"));
weight = new TextField(" ");

```

```

constT.add(weight);
constT.add(new Label("Name"));
constName = new TextField(" ");
constT.add(constName);

Panel rightPanel = new Panel();
rightPanel.setLayout(new GridLayout(3,1,5,10));
rightPanel.add(constP);
rightPanel.add(pageP);
rightPanel.add(constT);

Optimize.add("East",rightPanel);

Panel bottom = new Panel();
bottom.setLayout(new BorderLayout());
upDate = new Label("Load a hierarchy");
Button startOpt = new Button("Start optimization");
bottom.add("Center",upDate);
bottom.add("East", startOpt);
startOpt.addActionListener(this);
this.add("South",bottom);
this.add("East",Optimize);

this.setBackground( Color.white );
this.pack();
this.setVisible(true);
}

public void ReadInMFDData(String directory, String fileName)
{
    try
    {
        String path = directory+"/"+fileName;
        BufferedReader readIn = new BufferedReader(new FileReader(path));

        // Get name of MFD
        String s = readIn.readLine();
        StringTokenizer t = new StringTokenizer(s,"&");
        String Name = t.nextToken();
        this.Name = Name;
        this.setTitle("MFD: "+Name);

        // Get type of MFD interaction
        s = readIn.readLine();
        t = new StringTokenizer(s,"&");
        String Interaction = t.nextToken();

        // Get size of MFD in inches
        s = readIn.readLine();
        t = new StringTokenizer(s, " ");
        float MFDxSize = Float.valueOf(t.nextToken()).floatValue();
        float MFDySize = Float.valueOf(t.nextToken()).floatValue();

        // Get buttons
        s = readIn.readLine(); // skips line indicating buttons
        int count=0;
        Vector Buttons = new Vector();
        s=readIn.readLine();
        while(!s.equals("&&"))
        {
            // add to listing of buttons for constraints
            showButtons.addItem("b"+Buttons.size());

            t= new StringTokenizer(s, " ");
            float BxSize = Float.valueOf(t.nextToken()).floatValue();
            float BySize = Float.valueOf(t.nextToken()).floatValue();
            float BxPlace = Float.valueOf(t.nextToken()).floatValue();
            float ByPlace = Float.valueOf(t.nextToken()).floatValue();
            Buttons.addElement(new MFDBButton(""+count,BxSize,BySize,BxPlace,ByPlace));
            count++;
        }
    }
}

```

```

        s=readIn.readLine();
    }

    // all done!
    if(mfdframe!=null)
        mfdframe.dispose();
    mfdframe = new MFDFrame(this,Name,Interaction,MFDxSize,MFDySize,Buttons);
    ReadInHierarchyData(directory);
}
catch(Exception e)
{
    System.out.println("Error: "+e.toString());
}
}

public void ReadInHierarchyData(String cwd)
{
    Vector Pages = new Vector();
    File currentDir = new File(cwd);
    if (currentDir.isDirectory())
    {
        // read in file names
        String[] labels;
        FilenameFilter filter = new IgnoreLoadFiles("hdw", "mfd");
        labels = currentDir.list(filter); // read in all files, except .hdw and .mfd files
        // should be only one directory here, the top directory for the hierarchy
        for(int i=0;i<(int)Math.min(labels.length, mfdframe.Buttons.size());i++)
        {
            File labelFile = new File(currentDir+File.separator+labels[i]);
            MFDPage top = LoadLabelsForPage(labelFile, Pages);
            top.Parent = top;
        }
    }
    // Set up parent for each page
    for(int i=0;i<Pages.size();i++)
    {
        MFDPage currentPage = (MFDPage)Pages.elementAt(i);
        // find parent for this page
        for(int j=0;j<Pages.size();j++)
        {
            MFDPage checkPage = (MFDPage)Pages.elementAt(j);
            // see if currentPage is a sibling of this page
            if(checkPage.Siblings.contains(currentPage))
            {
                currentPage.Parent = checkPage;
                j=Pages.size();
            }
            // set default probability as a uniform distribution
            checkPage.Proportion = 1.0/Pages.size();
        }
    }
    mfdhierarchy = new MFDHierarchy(this,Pages);

    // List Pages unordered
    for(int i=0;i<Pages.size();i++)
    {
        MFDPage temp = (MFDPage)Pages.elementAt(i);
        String Name=temp.Name;
        if(temp.TypeOfPage==MFDPage.PARENT)
            Name = Name+" >";
        else if(temp.TypeOfPage==MFDPage.HYPERLINK)
        {
            Name = Name+" -<";
            upDate.setText("Some hyperlinks are not resolved.");
        }
        PageList.addItem(Name+"    p="+((float)temp.Proportion));
    }
    this.pack();
    newConstraint.enable();
    setVisible(true);
}

```

```

}

public MFDPage LoadLabelsForPage(File currentDir, Vector Pages)
{
    MFDPage returnPage=null;
    try
    {
        if(currentDir.isDirectory()) // parent
        {
            // read in file names
            String[] labels;
            FilenameFilter filter = new IgnoreLoadFiles("hdw", "mfd");
            labels = currentDir.list(filter); // read in all files, except .hdw and .mfd files from top
            // create siblings for this page
            Vector siblings = new Vector();
            for(int i=0;i<(int)Math.min(labels.length, mfdframe.Buttons.size());i++)
            {
                File labelFile = new File(currentDir+File.separator+labels[i]);
                MFDPage sibPage = LoadLabelsForPage(labelFile, Pages);
                sibPage.ButtonAssignment = i;
                siblings.addElement(sibPage);
            }

            // Create page
            returnPage = new MFDPage(currentDir.getName(), MFDPage.PARENT, siblings);
        }
        else // terminator (or hyperlink)
        {
            // read first line from file to see if it is a hyperlink
            try
            {
                BufferedReader readIn = new BufferedReader(new FileReader(currentDir));
                // Get type of page
                String s = readIn.readLine();
                if(s.equals("HYPERLINK"))
                {
                    returnPage = new MFDPage(currentDir.getName(), MFDPage.HYPERLINK);
                }
                else
                    returnPage = new MFDPage(currentDir.getName(), MFDPage.TERMINATOR);
            }
            catch(Exception e)
            {
                System.out.println("Error: "+e.toString());
            }
        }
        Pages.addElement(returnPage);
    }
    catch(Exception e)
    {
        System.out.println("Error: "+e.toString());
    }
    return(returnPage);
}

public static void main(String[] args)
{
    MFD f = new MFD("MFD");
    f.setVisible(true);
}

public void LoadPageProperties(MFDPage page)
{
    // Load up cell properties
    if(!LookingForHyperlink)
    {
        editingPage = page;
        changeProportion.setText(""+page.Proportion);
        checkFixed[1].setState(page.fixedProportion);
        checkFixed[0].setState(!page.fixedProportion);
    }
}

```

```

editLabel.setText("Edit proportion of time user targets this page: "+editingPage.Name);
if(page.TypeOfPage == MFDPPage.HYPERLINK)
{
    setHyperlink.enable();
    if(page.HyperLink == null)
        nameHyperlink.setText("No hyperlink set");
    else
        nameHyperlink.setText(page.HyperLink.Name);
}
else
{
    setHyperlink.disable();
    nameHyperlink.setText("No hyperlink required");
}
}
else
{
    nameHyperlink.setText(page.Name);
}
}

public void UpdateProportions()
{
    int size = mfdhierarchy.Pages.size();
    double fixedSum=0.0, freeSum=0.0;
    for(int i=0;i<size;i++)
    {
        MFDPPage tempPage = (MFDPPage)mfdhierarchy.Pages.elementAt(i);
        if(tempPage.fixedProportion)
            fixedSum += tempPage.Proportion;
        else
            freeSum += tempPage.Proportion;
    }
    for(int i=0;i<size;i++)
    {
        MFDPPage tempPage = (MFDPPage)mfdhierarchy.Pages.elementAt(i);
        if(!tempPage.fixedProportion)
            tempPage.Proportion = (1.0-fixedSum)*tempPage.Proportion/freeSum;
        String Name=tempPage.Name;
        if(tempPage.TypeOfPage==MFDPPage.PARENT)
            Name = Name+">";
        else if(tempPage.TypeOfPage==MFDPPage.HYPERLINK)
        {
            Name = Name+"-";
            upDate.setText("Some hyperlinks are not resolved.");
        }
        PageList.replaceItem(Name+" p="+((float)tempPage.Proportion,i);
    }
}

public void itemStateChanged(ItemEvent event)
{
    int selected = PageList.getSelectedIndex();
    if(selected!=-1)
    {
        MFDPPage temp = (MFDPPage)mfdhierarchy.Pages.elementAt(selected);
        mfdhierarchy.currentPage = temp;
        mfdhierarchy.ShowActivePage(temp);
        LoadPageProperties(temp);
        upDate.setText(" ");
    }
}

public void actionPerformed (ActionEvent event)
{
    String arg = event.getActionCommand();
    if (arg.equals("New"))
    {
        FileDialog d = new FileDialog(this, "Open MFD start", FileDialog.LOAD);
        d.setDirectory(lastDir);
    }
}

```

```

        d.show();
        String f = d.getFile();
        lastDir = d.getDirectory();
        if (f != null)
            ReadInMFDData(lastDir,f);
    }
    else if (arg.equals("Open"))
    {
        FileDialog d = new FileDialog(this, "Open MFD", FileDialog.LOAD);
        d.setDirectory(lastDir);
        d.show();
        String f = d.getFile();
        lastDir = d.getDirectory();
        if (f != null)
        {
            try
            {
                ObjectInputStream mfdStream = new ObjectInputStream(new FileInputStream(f));

                MFD newMFD = (MFD)mfdStream.readObject();
                newMFD.setVisible(true);

                float screenResolution = Toolkit.getDefaultToolkit().getScreenResolution();

                float xSize = newMFD.mfdframe.xSize;
                float ySize = newMFD.mfdframe.ySize;
                newMFD.mfdframe.setResizable(true);

                newMFD.mfdframe.setSize(((int)(screenResolution*xSize)+newMFD.mfdframe.getInsets().left+newMFD.mfdframe.getInsets().right,(int)(screenResolution*ySize)+newMFD.mfdframe.getInsets().top+newMFD.mfdframe.getInsets().bottom);

                newMFD.mfdframe.setResizable(false);
                newMFD.mfdframe.setVisible(true);
                newMFD.mfdhierarchy.setVisible(true);
            }
            catch(Exception e)
            {
                System.out.println("Error: "+e.toString());
            }
        }
    }
    else if (arg.equals("Save"))
    {
        FileDialog d = new FileDialog(this, "Save MFD", FileDialog.SAVE);
        d.setDirectory(lastDir);
        d.show();
        String f = d.getFile();
        lastDir = d.getDirectory();
        if (f != null)
        {
            try
            {
                ObjectOutputStream mfdStream = new ObjectOutputStream(new FileOutputStream(f));
                mfdStream.writeObject(this);
            }
            catch(Exception e)
            {
                System.out.println("Error: "+e.toString());
            }
        }
    }
    else if (arg.equals("Save changes to page"))
    {
        editingPage.Proportion = Float.valueOf(changeProportion.getText()).floatValue();
        editingPage.fixedProportion = checkFixed[1].getState();
        if(LookingForHyperlink)
            editingPage.HyperLink = mfdhierarchy.currentPage;
        LookingForHyperlink = false;
        // Calculate proportions
        UpdateProportions();
    }

```

```

}
else if (arg.equals("Change hyperlink"))
{
    LookingForHyperlink = true;
    nameHyperlink.setText("Select a page as the link from this page.");
}
else if (arg.equals("Start optimization"))
{
    MFDOptimize mfdoptimize = new MFDOptimize(this);
}
else if (arg.equals("New constraint")) // organize interface to get correct info
{
    currentConstraint = null;
    String choice = constraintChoice.getSelectedItemAt();
    saveConstraint.enable();
    weight.setText("1.0");
    constName.setText("C"+Constraints.size());
    constrainedPagesList.clear();
    int[] buttonIndex = showButtons.getSelectedIndexes();
    for(int i=0;i<buttonIndex.length;i++)
        showButtons.deselect(buttonIndex[i]);
    if(choice.equals("Global movement time"))
    {
        addPageConst.disable();
        upDate.setText("Global constraint, provide weight and (optional) name");
        ConstrainedPages = mfdhierarchy.Pages;
    }
    else
    {
        upDate.setText("Enter pages, buttons, weight and (optional) name");
        ConstrainedPages = new Vector();
        addPageConst.enable();
        deletePageConst.enable();
    }
}
else if (arg.equals("Add MFD page"))
{
    if(!ConstrainedPages.contains(mfdhierarchy.currentPage))
    {
        ConstrainedPages.addElement(mfdhierarchy.currentPage);
        constrainedPagesList.clear();
        for(int i=0;i<ConstrainedPages.size();i++)
            constrainedPagesList.addItem(((MFDPage)ConstrainedPages.elementAt(i)).Name);
        upDate.setText("Page constrained.");
    }
    else
        upDate.setText("Page already constrained.");
}
else if (arg.equals("Remove selected page"))
{
    int index = constrainedPagesList.getSelectedIndex();
    if(index!= -1)
    {
        ConstrainedPages.removeElementAt(index);
        constrainedPagesList.clear();
        for(int i=0;i<ConstrainedPages.size();i++)
            constrainedPagesList.addItem(((MFDPage)ConstrainedPages.elementAt(i)).Name);
        upDate.setText("Page removed from constraint.");
    }
    else
        upDate.setText("Select a page to remove.");
}
else if (arg.equals("Save constraint"))
{
    saveConstraint.disable();
    addPageConst.disable();
    deletePageConst.disable();
    // get selected buttons
    int[] buttonIndex = showButtons.getSelectedIndexes();
    // get weight

```

```

double tempWeight = (double)Float.valueOf(weight.getText()).floatValue();
// get name
String tempName = constName.getText();
String choice = constraintChoice.getSelectedItemId();
// only Fixed place needs button info
if(choice.equals("Pages to fixed buttons"))
{
    if(buttonIndex.length>0)
    {
        MFDCConstraint tempConst = new MFDCConstraint(tempName, ConstrainedPages,
MFDCConstraint.RESTRICTEDPLACES, tempWeight, buttonIndex);
        if(currentConstraint==null)
        {
            Constraints.addElement(tempConst);
            upDate.setText("Constraint added");
        }
        else
        {
            int ccIndex = Constraints.indexOf(currentConstraint);
            Constraints.setElementAt(tempConst,ccIndex);
            upDate.setText("Constraint updated");
        }
    }
    else
        upDate.setText("Select buttons");
}
else if (choice.equals("Global movement time"))
{
    MFDCConstraint tempConst = new MFDCConstraint(tempName, ConstrainedPages, MFDCConstraint.MOVEMENTTIME,
tempWeight);
    if(currentConstraint==null)
    {
        Constraints.addElement(tempConst);
        upDate.setText("Constraint added");
    }
    else
    {
        int ccIndex = Constraints.indexOf(currentConstraint);
        Constraints.setElementAt(tempConst,ccIndex);
        upDate.setText("Constraint updated");
    }
}
else if (choice.equals("Path movement time"))
{
    MFDCConstraint tempConst = new MFDCConstraint(tempName, ConstrainedPages, MFDCConstraint.PATHMOVEMENTTIME,
tempWeight);
    if(currentConstraint==null)
    {
        Constraints.addElement(tempConst);
        upDate.setText("Constraint added");
    }
    else
    {
        int ccIndex = Constraints.indexOf(currentConstraint);
        Constraints.setElementAt(tempConst,ccIndex);
        upDate.setText("Constraint updated");
    }
}
else if (choice.equals("Pages to close buttons"))
{
    MFDCConstraint tempConst = new MFDCConstraint(tempName, ConstrainedPages, MFDCConstraint.RELATEDNEARBY,
tempWeight);
    if(currentConstraint==null)
    {
        Constraints.addElement(tempConst);
        upDate.setText("Constraint added");
    }
    else
    {
        int ccIndex = Constraints.indexOf(currentConstraint);

```

```

        Constraints.setElementAt(tempConst,ccIndex);
        upDate.setText("Constraint updated");
    }
}
showConstraints.clear();
for(int i=0;i<Constraints.size();i++)
    showConstraints.addItem((MFDConstraint)Constraints.elementAt(i)).Name);
currentConstraint = null;
}
else if (arg.equals("Delete constraint"))
{
    int index = showConstraints.getSelectedIndex();
    if(index!= -1)
    {
        Constraints.removeElementAt(index);
        showConstraints.clear();
        for(int i=0;i<Constraints.size();i++)
            showConstraints.addItem((MFDConstraint)Constraints.elementAt(i)).Name);
        upDate.setText("Constraint deleted");
    }
    else
        upDate.setText("Select a constraint first");
}
else if (arg.equals("Edit constraint"))
{
    int index = showConstraints.getSelectedIndex();
    if(index!= -1)
    {
        currentConstraint = (MFDConstraint) Constraints.elementAt(index);
        weight.setText(""+currentConstraint.Weight);
        constName.setText(currentConstraint.Name);
        constraintChoice.select(currentConstraint.TypeOfConstraint);
        ConstrainedPages = currentConstraint.Pages;
        // load up constrained pages
        constrainedPagesList.clear();
        for(int i=0;i<ConstrainedPages.size();i++)
            constrainedPagesList.addItem((MFDPPage)ConstrainedPages.elementAt(i)).Name);

        int[] buttonIndex = showButtons.getSelectedIndexes();
        for(int i=0;i<buttonIndex.length;i++)
            showButtons.deselect(buttonIndex[i]);

        if(currentConstraint.TypeOfConstraint == MFDConstraint.RESTRICTEDPLACES)
        {
            for(int i=0;i<currentConstraint.Places.length;i++)
                showButtons.select(currentConstraint.Places[i]);
        }
        addPageConst.enable();
        deletePageConst.enable();
        saveConstraint.enable();
        upDate.setText("Constraint opened");
    }
    else
        upDate.setText("Select a constraint first");
}
}
}

```

```

// File filter to ignore .mfd and hdw files
class IgnoreLoadFiles implements FilenameFilter
{
    private String extension1, extension2;
    public IgnoreLoadFiles(String extension1, String extension2)
    {
        this.extension1 = extension1;
        this.extension2 = extension2;
    }
    public boolean accept(File dir, String name)
    {
        if(!name.endsWith(extension1) && !name.endsWith(extension2))

```

```
        return true;
    else
        return(new File(dir,name)).isDirectory();
    }
}
```

Appendix B.

MFDTool user's guide.

The user's guide provides detailed information on how to use MFDTool and discusses an example of MFD design.

MFDTool: A software aid for the design of multifunction displays

User's Guide

Gregory Francis¹
Purdue University
1364 Psychological Sciences Building
West Lafayette, IN 47907
gfrancis@psych.purdue.edu
<http://www.psych.purdue.edu/~gfrancis/home.html>
and
U.S. Army Aeromedical Research Laboratory
Ft. Rucker, AL 36362-0577

August 11, 1999

¹The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

This work was supported by the Army Aeromedical Research Laboratory (Kent Kimball, Ph.D.) under the auspices of the U.S. Army Research Office Scientific Services Program administered by Battelle (Delivery Order 398, Contract No. DAAHO4-96-C-0086).

Abstract

This document is a guide to MFDTTool, a software aid for the design of multifunction displays (MFDs). MFDTTool applies an optimization algorithm to designer-specified constraints thereby creating the best layout of MFD information for MFD hardware. The guide specifies the types of MFD situations where MFDTTool applies and describes the steps needed to define constraints and start the optimization approach. A sample MFD design problem (involving an automated teller machine) is discussed.

Contents

1 Purpose	2
1.1 Some difficulties of MFD design	2
1.2 How MFDTool helps	3
1.3 Constraint costs	4
1.3.1 Global movement time	5
1.3.2 Pages to close buttons	6
1.3.3 Pages to fixed buttons	6
1.3.4 Path movement time	7
1.4 Weighting costs	8
1.5 Optimization	8
2 Using MFDTool	10
2.1 Running MFDTool	10
2.2 Creating MFD files	10
2.2.1 MFD hardware	10
2.2.2 MFD pages	14
2.3 Setting constraints	18
2.3.1 Global movement time	18
2.3.2 Pages to close buttons	22
2.3.3 Pages to fixed buttons	22
2.3.4 Path movement time	24
2.4 Optimizing	25
2.5 Saving MFDs	27
3 Conclusions	27

1 Purpose

MFDTool is software that helps a MFD designer optimize assignment of MFD information to MFD hardware commands (e.g., button pushes). To understand how MFDTool can help in the design process; the following section describes some of the tasks involved in MFD design.

1.1 Some difficulties of MFD design

Many computer devices for information display involve fixed hardware switches and flexible software pages. These types of devices are called multifunction displays (MFDs) or multi-purpose displays. The information in these devices is often arranged hierarchically so that a user starts at a top level and moves down the hierarchy by selecting appropriate MFD pages. Often the hardware devices are real or simulated buttons that remain in fixed positions. Common examples of MFDs include automated teller machines, pagers, aircraft cockpit display panels, and various medical devices. As the user moves through the hierarchy, the MFD screen changes, thereby providing information to the user. The creation of effective MFDs is a difficult task, and designers must decide how many buttons to include, the hierarchical arrangement of information, and the mapping of page labels to hardware (typically buttons). MFDTool is a computer program that helps in the last of these tasks.

Before discussing what MFDTool does, it may be useful to discuss what it does *not* do. MFDTool cannot specify MFD hardware. Decisions on MFD hardware are influenced by a variety of factors, including environmental conditions, cost of devices, previous versions of the MFD, and other details that may be particular to a MFD's specific purpose. In general, there is no method for *a priori* designing the MFD hardware, although there are a variety of guidelines. Often times a designer is simply given a standard hardware setup and told to put the information "in" the device. MFDTool also does not select what types of information should be included in the MFD. The selection of information in a MFD and its hierarchical arrangement is a task that largely must be determined by a human designer because it involves identifying what types of information are necessary, and creating labels that are meaningful for the user and the MFD purpose. No computer or algorithm, currently, can identify the semantic and organizational relationships between MFD information.

However, at some point in the design of a MFD, decisions must be made about how to map the various parts of the information hierarchy to user actions (e.g., button pushes). Mapping hierarchy information to MFD buttons is a challenging task. The human-computer interactions involved in accessing information from a MFD are complicated and not entirely understood. Moreover, even a small hierarchy database can be mapped to hardware buttons in a vast number of ways, so combinatorial explosion quickly precludes an exhaustive search of all possible mappings. Therefore such mappings are, at best, created by experts who rely on experience and general guidelines like the following:

1. Frequently used functions should be the most accessible.
2. Time critical functions should be the most accessible.

3. Frequently used and time critical functions should be activated by the buttons that feel “ideally located” (e.g., top of a column of buttons).
4. Program repeated selection of the same button. Failing that, program sequential selections to adjacent buttons.
5. The number of levels in the hierarchy should be as small as possible.
6. The overall time to reach functions should be minimized.
7. Functions that are used together should be grouped on the same or adjacent pages.
8. Related functions on separate pages should be in a consistent location.
9. Related functions should be listed next to each other when on a single page.
10. Consider the types of errors crew members might make and place functions accordingly to minimize the effect of those errors.

While many of these guidelines correctly identify the key characteristics of good MFD design, application of these criteria is problematic because they often conflict with each other. For example, should a frequently used function be placed by itself near the top of the hierarchy (1) or should it be placed in a submenu with its related, but infrequently used, functions (7)? Likewise, should criteria (3), (4) or (7) dominate selection of a button for a specific function? Currently, there is no quantitative method of measuring the tradeoffs and designers try out different options until the whole system “feels” good. This is a time consuming task because movement of a single function can require other changes throughout the MFD. As a result, hierarchy creation largely remains an artistic endeavor, depending primarily on the experience, intuition, and hard work of the designer.

The best artists use good tools to help them in their craft. MFDTool handles part of the complexity of measuring the impact of various guidelines. MFDTool cannot build a MFD from scratch, and the design will still depend on the experience and effort of the designer. MFDTool does allow the designer to consider a larger range of possibilities by automating part of the design process, thereby freeing the designer to focus on other tasks.

1.2 How MFDTool helps

MFDTool focuses on a subset of the guidelines identified above that can be recast in terms of an optimization problem. MFDTool requires that the designer has a specified MFD hardware system that describes the sizes and positions of MFD buttons (the approach can be modified to other types of interactions, but buttons are a common interface type). MFDTool also requires that the designer specifies the hierarchical arrangement of information pages in the database. This arrangement also allows for hyperlinks that move back up the hierarchy (e.g., RETURN) or function as shortcuts, as described below.

Given this information, MFDTool allows the user to identify four types of constraints, which can be mixed and matched as desired.

1. **Global movement time:** If one ignores shortcuts and backward links, then moving through the hierarchy from top to bottom can be described by a single finite sequence of button presses. When the designer specifies the frequencies of search for different pieces of MFD information, MFDTool associates page labels with buttons in a way to minimize the average movement time needed to reach information. This constraint corresponds to guidelines (1), (6), and often (4), above.
2. **Pages to close buttons:** Often labels on a single screen are related to each other and the designer wants the related page labels to be grouped together on nearby buttons. At other times labels on different MFD screens are related and the designer wants those labels to be associated to the same or nearby buttons (e.g., CANCEL should be in the same place on every page). MFDTool allows the designer to specify as many of these constraints as desired. This constraint corresponds to guidelines (8), and (9), above.
3. **Pages to fixed buttons:** Sometimes a designer wants to restrict a single label or multiple labels (either on the same screen or different screens) to a subset of the possible buttons (e.g., always put left engine information on the left side of the MFD screen). MFDTool allows the designer to specify as many of these constraints as desired. This constraint accommodates guideline (3) above, but also allows for more general restrictions.
4. **Path movement time:** The use of some MFDs requires users to retrieve certain combinations of information. If a user has to first check the status of one system, then the status of a second, and then the status of a third, there will be a path of visited pages that correspond to this combination of information searches. Moreover, because the system information may be scattered across the MFD hierarchy, designers often include hyperlinks, or shortcuts, to the top of the hierarchy or to other MFD hierarchy locations. MFDTool allows the designer to identify these paths and acts to assign page labels to buttons to minimize the time required to execute these sequences. MFDTool allows the designer to specify as many of these paths as desired.

In MFDTool, each constraint has a corresponding numerical cost function that measures how poorly a constraint is being satisfied by the current MFD design. Larger cost values correspond to worse designs. An optimization algorithm searches through a variety of MFD designs to find one that minimizes (or nearly so) the sum of costs. The calculation of costs is described in the next section. The next section is fairly complex, involving a mathematical description of costs, and it can be skimmed by beginners who want to learn how to use the methods of MFDTool, but are not yet interested in the underlying principles.

1.3 Constraint costs

There are four types of constraints, as described in the previous section. MFDTool acts to minimize cost functions associated with these constraints. This section mathematically defines the cost functions.

1.3.1 Global movement time

In MFDTool, insuring that needed information can be retrieved as quickly as possible corresponds to placing MFD labels on buttons that minimize the time needed to execute the movements. For a user new to the use of a particular MFD, the savings of such minimization may be small, as much of the searching time involves reading labels and identify which buttons to press. However, for an expert user, most of the search time consists of executing the already known sequences of button presses. Identifying which button press sequences are fastest and assigning frequently searched for items to those button press sequences can lead to substantial reductions in access time.

Applying this approach requires a means of predicting how long it will take an expert user to execute a sequence of button presses. MFDs can be used with a variety of interactions (e.g., mouse clicks, finger-pointing, multiple-finger movements, special pointer pens, step cursor control, hand-on-throttle). Models for different types of interactions are dramatically different. At the moment, MFDTool supports only finger-pointing movements because there is a well established model of how long it takes people to move a pointer over a given distance to a target of a given size. MFDTool uses a form of Fitts' Law that says that the movement time, M is:

$$M = I_m \log_2 \left(\frac{2D}{S} + 1 \right). \quad (1)$$

Here, D is the distance between the starting position of the finger and the target; S is the size of the target (MFDTool measures this as the minimum of button height and width), \log_2 is the logarithm in base 2, and I_m is a parameter with units milliseconds/bit. I_m is empirically measured, and, for finger movements, values between 70 and 120 ms/bit are common. MFDTool uses $I_m = 100$ ms/bit.

When a sequence of movements is to be executed, MFDTool makes the simplifying assumption that it can add up the M terms for movement from the first button to the second, the second to the third, and so on. Thus, the total time needed to execute a sequence of button presses will be:

$$T = \sum_{i=1}^{m-1} M_{i,i+1}, \quad (2)$$

where there are m button presses in the sequence, and $M_{i,i+1}$ is the time to move between successive buttons. This is almost surely a lower limit of execution time, as a user may need to read labels to remember which button to press next. There is no *a priori* way to know when a user will memorize the pattern of button presses to retrieve particular information. Such memorization surely depends on the semantics of the hierarchy and the user's experience. MFDTool has no way to model these effects.

Future versions of MFDTool will include support for other types of interactions, including movement with a mouse, pointer pens, step cursor control, and hand-on-throttle control. A more difficult task is to model multiple-finger movements (e.g., typing or piano playing), though it may be possible in certain situations.

Once the interaction model is defined, MFDTool can predict how long it will take to reach a desired information label by looking at the sequence of button pushes necessary to

reach that page label from the top level of the MFD hierarchy. The cost function for global movement time is the average time to reach a MFD page label:

$$C_1 = \sum_{j=1}^n T_j p_j, \quad (3)$$

where n is the number of information labels in the hierarchy, T_j is the total time needed to execute the sequence of button presses to reach page label j , and p_j is the proportion of time that page label j is needed by the user. As the assignment of page labels to buttons is modified, the value of T_j changes. MFDTool tries to assign page labels to button presses so that labels with larger p_j values have smaller T_j values, thereby minimizing search time.

1.3.2 Pages to close buttons

One could imagine a situation where a user is very knowledgeable about searching through a MFD and has memorized all the button presses to reach every page label. In such a situation, the best the designer can do is to minimize the total movement time using C_1 . However, such situations are rare. Even experienced users probably use feedback from the MFD to guide their searches for all but the most commonly used page labels. As a result, the designer needs to provide order among the assignment of labels to buttons that will help guide the user's search. A commonly used technique is to place labels that are related to each other on nearby buttons. A designer may, for example, want to create ordered lists of items on a single MFD screen and may also want to insure that related labels on different screens are associated with nearby buttons.

In its present version, MFDTool defines "closeness" relative to the time needed to move between buttons. Thus, if a designer constrains page labels L_1, \dots, L_k to be as close as possible, and each label is currently assigned to buttons $b(L_1), \dots, b(L_k)$, then the quantitative cost of these assignments is:

$$C_2 = \sum_{i=1}^k \sum_{j=i+1}^k M[b(L_i), b(L_j)]. \quad (4)$$

Here $M[b(L_i), b(L_j)]$ is the time needed to move from button $b(L_i)$ to button $b(L_j)$, as in equation (1). The second summation starts at $i + 1$ to avoid double summation of time for each button pair. If the labels in this constraint are all on different pages, then C_2 equals zero when every label is associated with the same button. If some of the labels are on the same screen, C_2 has a nonzero minimum, as two labels cannot simultaneously be associated with the same button on the same screen. Whichever the case, MFDTool tries to assign information labels to buttons in a way that minimizes C_2 .

1.3.3 Pages to fixed buttons

Sometimes a designer may want to constrain some page labels to a particular button or set of buttons. This could occur for example, if a designer, to stay consistent with other displays, wants an EXIT label always placed on the lower left button. Or, a designer may

want geographical topics to have corresponding positions on the MFD screen (e.g., left to left). All of these constraints can be imposed in MFDTool.

This constraint cost measures how close the page labels are to their restricted buttons. As with the other costs, MFDTool defines “closeness” relative to the time needed to move between buttons. Thus, if the designer constrains page labels L_1, \dots, L_k to be restricted to buttons b_1, \dots, b_h , and each label is currently assigned to buttons $b(L_1), \dots, b(L_k)$, then the quantitative cost of these assignments is:

$$C_3 = \sum_{i=1}^k \min_{j=1, \dots, h} M[b_j, b(L_i)]. \quad (5)$$

The term inside the summation compares the currently assigned button for label L_i with each of the allowable buttons and takes the minimum movement time. Thus, if all labels are assigned to one of the allowable buttons, the minimum movement times are zero and the total cost is zero. When a constrained label is not assigned to an allowable button, the cost is incremented by the minimum movement time needed to move from the assigned button to one of the allowable buttons.

1.3.4 Path movement time

C_1 , above, measures the average time required to search for a page label, starting from the top of the hierarchy and taking the most direct route to that label. However, depending on the MFD, not all searches are of that type. It is frequently the case that a user needs to gather a number of different types of information from different screens in the MFD. The designer may include shortcuts or hyperlinks that allow the user to quickly travel along such paths of pages. Cost C_1 cannot account for these types of situations because the use of shortcuts means that there are multiple (usually infinitely many) ways to reach a label. For these types of situations, the designer must specify the sequence, or path, of pages the user goes through to perform a required task. Once this path is specified, MFDTool acts to minimize movement time along that path by associating page labels to buttons, much as for cost C_1 .

The quantitative definition of cost is much as for C_1 , except the designer must identify the path of page labels that the user steps through (for C_1 the computer could do this because each page label has a unique position in the hierarchy). The designer identifies an ordered sequence of page labels L_1, \dots, L_k for which movement time is to be minimized. If each page label is currently assigned to buttons $b(L_1), \dots, b(L_k)$, then the quantitative cost of these assignments is:

$$C_4 = \sum_{i=1}^{k-1} M[b(L_i), b(L_{i+1})]. \quad (6)$$

MFDTool tries to minimize this cost through the assignment of page labels to buttons.

In some MFD applications, minimization of movement time along these paths may be the most important job for the designer. By their very nature, such sequences must be specified by the designer.

1.4 Weighting costs

All of the constraint costs are defined in terms of milliseconds of time needed to move between buttons. However, the designer still needs to identify the relative importance of different constraints so that MFDDTool produces the desired result. It is common for constraints to be in conflict with each other. In anticipation of such conflicts the designer needs to indicate a weight, λ , for each constraint cost. For example, if the designer wants to be certain that the EXIT label is always on the lower left button, even if such assignment means an increase in average search time, then the weight for the EXIT constraint might be set larger than the weight for the average search time.

MFDDTool tries to minimize the weighted sum of constraint costs:

$$C = \sum_{i=1}^n \lambda_i R_i. \quad (7)$$

Here, there are n constraints defined by the user, and R_i corresponds to the cost associated with constraint i .

There is no way for MFDDTool to advise the designer on how to set the weights. The default is the value one, but it is merely a starting point and not intended as a reasonable choice. The values of the weights have a great effect on the resulting MFD design, and it is not unusual for a designer to tweak the weights to insure that one constraint is satisfied over another. The use of extremely large weights, relative to others, is often not effective because it sometimes hinders the optimization process (next section). In general, the designer should set the weight on a constraint just high enough to insure that the constraint is satisfied. This often involves trial and error.

1.5 Optimization

Once a total cost function is defined, one can use any number of algorithms to find the assignment of page labels to buttons that minimizes that cost function. MFDDTool currently uses the simulated annealing algorithm, but future versions of MFDDTool may explore other approaches.

Simulated annealing is a variation of hill-climbing algorithms. In a hill-climbing (or hill-descending, only the sign needs to be changed) algorithm, the system is initialized to a particular state (e.g., mapping of labels to buttons) and the cost is calculated for that state. One of the variables of the problem (e.g., a label) is randomly selected and modified (e.g., moved to a new button). A new cost value is calculated, and if the new cost is less than the old cost, the change is kept, otherwise the change is undone. In this way, the system converges to a state where any change would lead to an increase in cost (e.g., where any change in the mapping would be worse). Hill-climbing techniques have a tendency to get stuck in local minima of cost because they never accept changes that increase cost. In complex problems, hill-climbing methods can easily get trapped in a state where any change only increases cost but the global minimum is very different, with a much smaller cost. What is needed is a controlled way to climb out of local minima and end in a state with the global

minimum of cost. By analogy, one would probably, at some point during a hike, need to go down a ravine or a small slope to climb to the top of a mountain.

Simulated annealing is a stochastic algorithm that at first accepts changes even if they lead to larger costs. As time progresses a temperature parameter gradually decreases (this is the annealing) so that it becomes less likely that a change leading to an increase in cost is accepted. As the temperature becomes small the algorithm becomes essentially hill-climbing. As long as the temperature decreases slowly enough and enough changes are considered at each temperature level, simulated annealing is statistically guaranteed to find the global minimum of a problem. In practice, though, the necessary temperature schedule is too slow and the number of changes at each level is too big, so simpler approaches are taken that are faster, but less certain to find the global minimum.

In simulated annealing, the initial temperature, T , is set large enough that many state changes are accepted even if they lead to a cost increase. *MFDTool* sets the initial temperature in the following way. Given the state of the system at the start of the optimization process, many (50 times the number of page labels) changes are made to the MFD, and the change in cost is calculated for each change. The average of these cost changes is the initial temperature for the annealing process. The final state of the system after all these changes is also the initial state for the start of the annealing process.

Changes are made by randomly selecting a MFD page label. Its button assignment is noted, and a new button assignment is randomly selected. The selected label swaps positions with whatever (perhaps nothing) is at the new button assignment. After each change, new cost, C_{new} , is calculated and compared to the cost before the change, C_{old} . The change in cost, $\Delta C = C_{\text{new}} - C_{\text{old}}$, is calculated. If the cost change is negative, the change is kept. If the cost change is positive, the change is kept when a random number between zero and one is greater than

$$p = \exp(-\Delta C/T). \quad (8)$$

This relationship means that when ΔC is much smaller than T , p is close to one, and lots of changes are kept. As T gets smaller than ΔC , p gets closer to zero, and changes are not kept very often. Statistically then, the system is more likely to be in a state with a low cost. As T decreases, the system tends to be stuck in a state with very low cost.

To insure that the statistical situation is close to reality, one needs to implement many changes at every temperature level. *MFDTool* makes 300 times the number of page labels changes at every temperature level. After these changes, the temperature is modified by the equation

$$T_{\text{new}} = 0.99T_{\text{old}}. \quad (9)$$

The process is then repeated for the new temperature. The whole process stops when it seems that the temperature is so small that the system is trapped in a particular state (as in hill-climbing). *MFDTool* reaches this conclusion when ten changes in temperature have not produced any changes in cost.

At the end of the simulated annealing process, the system should be in a state with a low (but perhaps not optimal) cost. Being certain of finding the true optimal state with the absolute lowest possible cost would be prohibitively difficult and would likely require a supercomputer, even for relatively small MFDs.

2 Using MFDDTool

This section discusses how a designer uses MFDDTool to aid in the development of a MFD. Step-by-step instructions are provided, and an example involving the design of an automated teller machine (ATM) MFD is discussed in detail.

2.1 Running MFDDTool

You can download the most recent version of MFDDTool from <http://www.psych.purdue.edu/~gfrancis/MFDDTool/index.html>. The most recent version of this user's guide will also be at the same web address. There is no installation procedure, simply put the files in a directory of your choosing.

MFDDTool is written in Java, so it will run on any computer operating system that has a working Java virtual machine (VM). Java VMs are available for nearly every modern operating system; so in principle, MFDDTool should run almost anywhere. MFDDTool does not come with a Java VM, but one can be freely downloaded from <http://www.javasoft.com>. You will need to download either the Java Development Kit (JDK) or the Java Runtime Environment (JRE). The JDK is necessary if you intend to modify and recompile the Java source code. The JRE will allow you to run the compiled code. (Although many web browsers include a Java VM, they cannot run MFDDTool. MFDDTool must read from and write to the computer's hard drive, and the Java VM in web browsers prohibits such actions.)

Different operating systems integrate Java in different ways. With Microsoft Windows 95/98/NT, a Java program is started in a DOS window. In a DOS window go to the directory containing the MFDDTool files and type `java mfd` and then hit the return key. A window should appear like that in Figure 1, although it will vary somewhat from one operating system to another.

2.2 Creating MFD files

MFDDTool needs three basic types of information: a description of the MFD hardware components, a description of the hierarchical arrangement of information, and designer-specified constraints on how the information should be mapped to the hardware components.

2.2.1 MFD hardware

The hardware description includes the number of buttons, their size, and their spatial arrangement. This information is provided by the designer in a single text file. Figure 2 shows how the MFD hardware information is coded into a format for MFDDTool to understand. The first line of the file contains the name of the MFD, which in this case is ATM. An ampersand (&) follows the name, and any comments can be placed after the ampersand without being read by MFDDTool. The second line indicates the type of interaction possible for the user to interface with the MFD. In the current version of MFDDTool, this line is irrelevant as only finger-pointing is supported. The line needs to be present (with the ampersand at the end of the interaction type and optional comments following), but the contents currently

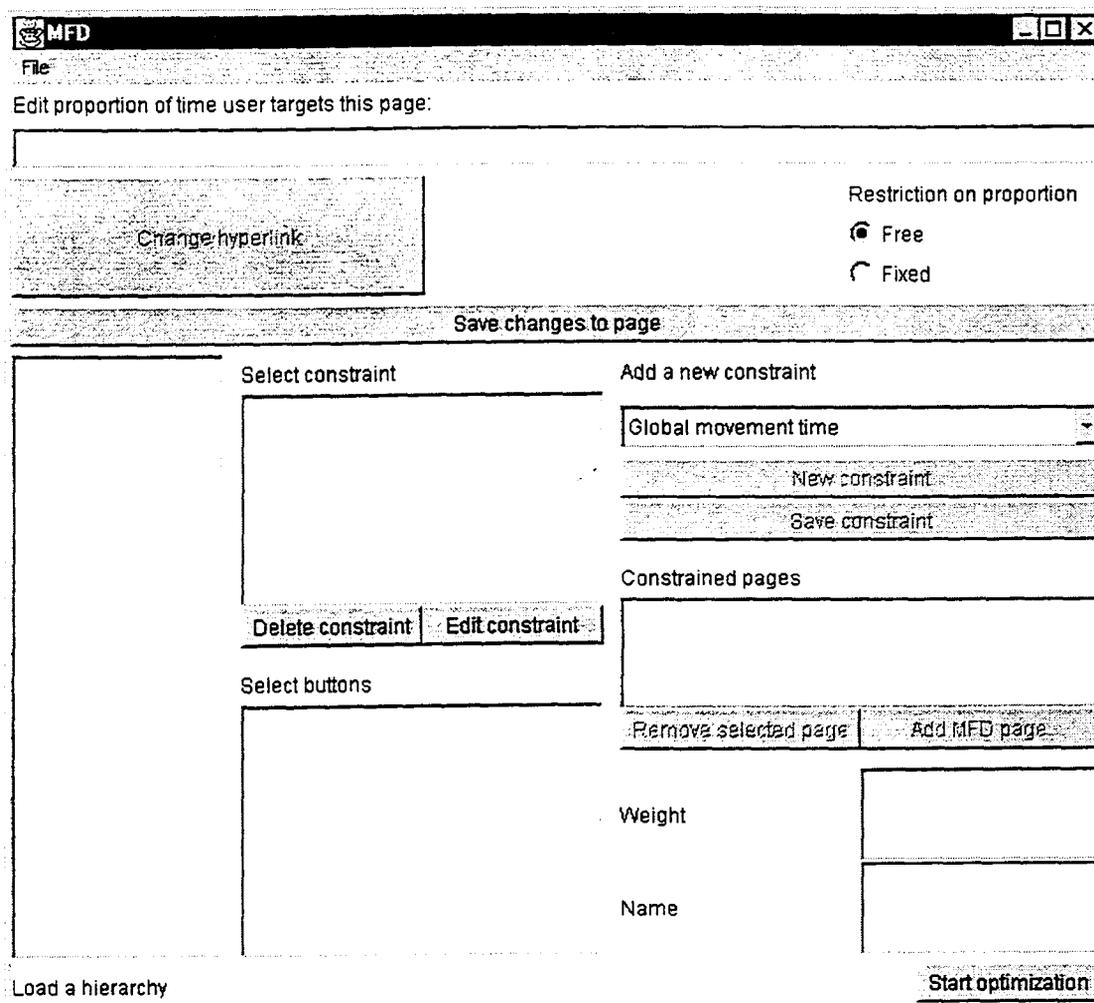


Figure 1: The MFDTool startup window, as seen in Windows 95.

MFDTool

```
ATM & Name
Hand & Interaction
6 6 & xSize, ySize
% Buttons
0.5 0.5 0.1 0.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 0.1 1.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 0.1 2.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 0.1 3.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 0.1 4.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 5.4 0.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 5.4 1.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 5.4 2.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 5.4 3.25 & xSize, ySize, xPlace, yPlace
0.5 0.5 5.4 4.25 & xSize, ySize, xPlace, yPlace
&&
```

Figure 2: Contents of the file ATM.hdw, which defines the size of the MFD and the physical properties of the buttons for the ATM example.

have no effect on MFDTool. In subsequent versions, MFDTool will distinguish between finger-pointing, mouse movements, cursor tabbing, etc.

The third line contains information on the physical dimensions of the MFD, again ending with an ampersand, which can optionally be followed by comments. The first number on the line is the x -dimension (horizontal width) of the MFD in inches. The second number on the line (the numbers are separated by a single blank space) is the y -dimension (vertical height) of the MFD in inches.

The fourth line is not read by MFDTool, but acts as a visual separator for the designer between the earlier information and information about MFD buttons. Each remaining line specifies details about the size and position of a button on the MFD. As the comments to the right of the ampersand on each line indicate, the first number on each line is the horizontal width of the button and the second line is the vertical height of the button. The third and fourth numbers are the x and y positions of the upper left corner of the button, as measured from the upper left corner of the MFD frame. All numbers are measured in inches. The very last line contains only two ampersands, which tell MFDTool that there are no more buttons.

From the MFDTool start window select the File menu, and select New. An open file window should appear. Go to the directory ATM and open the file ATM.hdw. Two new windows will appear, one titled MFD Hardware: ATM and another titled Hierarchy: ATM. Ignore the latter for the moment.

Figure 3 shows what the window titled MFD Hardware: ATM should look like on Windows 95. The size of the window is (roughly) six by six inches, and the size and placement of buttons should correspond to the details specified in the file ATM.hdw. (Even if the size and placement are off slightly, rest assured that the information used by MFDTool is exactly as specified in the file. Sometimes Java windows and buttons are created slightly different than as specified. The calculations of distance and sizes will be based on the designer-supplied numbers and not on how the MFD Hardware window looks.)

Notice that the two top left buttons have text (OK Password and Cancel) on them. This text corresponds to page labels that are currently associated with these buttons. Not all of

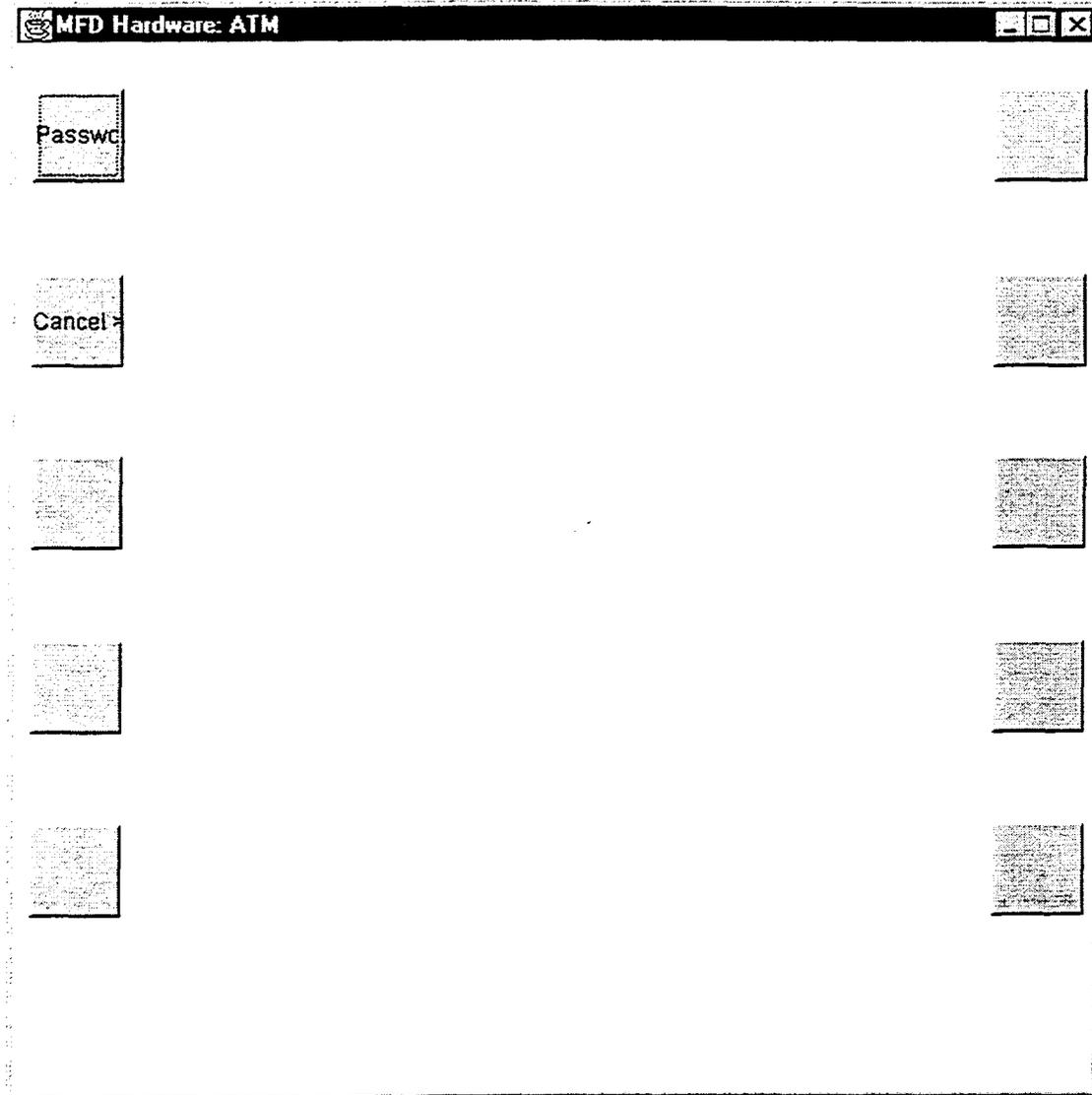


Figure 3: The MFD Hardware window shows the physical layout of the MFD and its buttons, as specified in the file ATM.hdw.

MFDTool

the text may be visible on a button, but this should be of no concern. MFDTool does not try to exactly emulate what the MFD hardware will look like. Many devices place the text to one side of buttons, while others place the text directly on the buttons. MFDTool does not distinguish between these display types. It only cares about the association between the label and the button. Likewise, MFDTool does not consider whether label text will fit on a button or next to a button. These types of concerns must be handled by the designer by choosing appropriate text and font sizes. More generally, the actual text of the labels is for the benefit of the designer (and ultimately the user), but MFDTool makes no use of the text itself.

Clicking on a button with a label reconfigures the MFD hardware window to show the contents of the page associated with that label. After selecting OK Password, the MFD Hardware window will look like the screenshot in Figure 4, with new labels on the buttons. The click “moved” the display screen down one level in the hierarchy to show the contents of the OK Password page. You can continue to select labels to move further down the hierarchy. However, you cannot yet move back up the hierarchy because any necessary hyperlinks that would move up the hierarchy have not been specified (below).

The MFD Hardware window is designed to give an idea of the current assignment of information labels to buttons. This allows the designer to determine whether the assignment is consistent with what was intended.

2.2.2 MFD pages

The text of labels and their hierarchical arrangement must be provided by the designer. This is done by creating a directory structure using your operating system’s file manager to create directories and files. MFDTool reads in this file structure and interprets the names of the directories and files as text labels for the MFD.

Using your operating system’s file browser open the MFDTool folder. Inside is a directory titled ATM. Open that directory, inside is the file ATM.hdw, which was discussed above. There is also a directory called Start. The Start directory is the top page of the MFD hierarchy (it can be called anything, but there should be only one directory with the *.hdw file). The Start directory contains two directories titled Cancel and OK Password, which correspond to the text labels in Figure 3. The OK Password directory contains additional directories and files that correspond to the text labels in Figure 4.

The set of page labels for the ATM example is defined as the directories and files under the Start directory, with the MFD hierarchy matching the file hierarchy. To create your own set of MFD information, simply create a directory and place a *.hdw file in it (with the specified description of the physical aspects of the MFD) and then create a directory structure of the information. MFDTool reads in the information to create *pages* and *labels* for those pages. The only restriction on page names is that files should not end in the extension .hdw or .mfd, as MFDTool reserves those extensions for the description of the MFD hardware (as discussed above) and for saving an entire MFD (as discussed below). Some operating systems also place restrictions on directory and file names (e.g., disallowing blank spaces or special characters). When reading the contents of a directory from the file system, MFDTool reads in only as many files as can possibly be assigned to buttons. In such

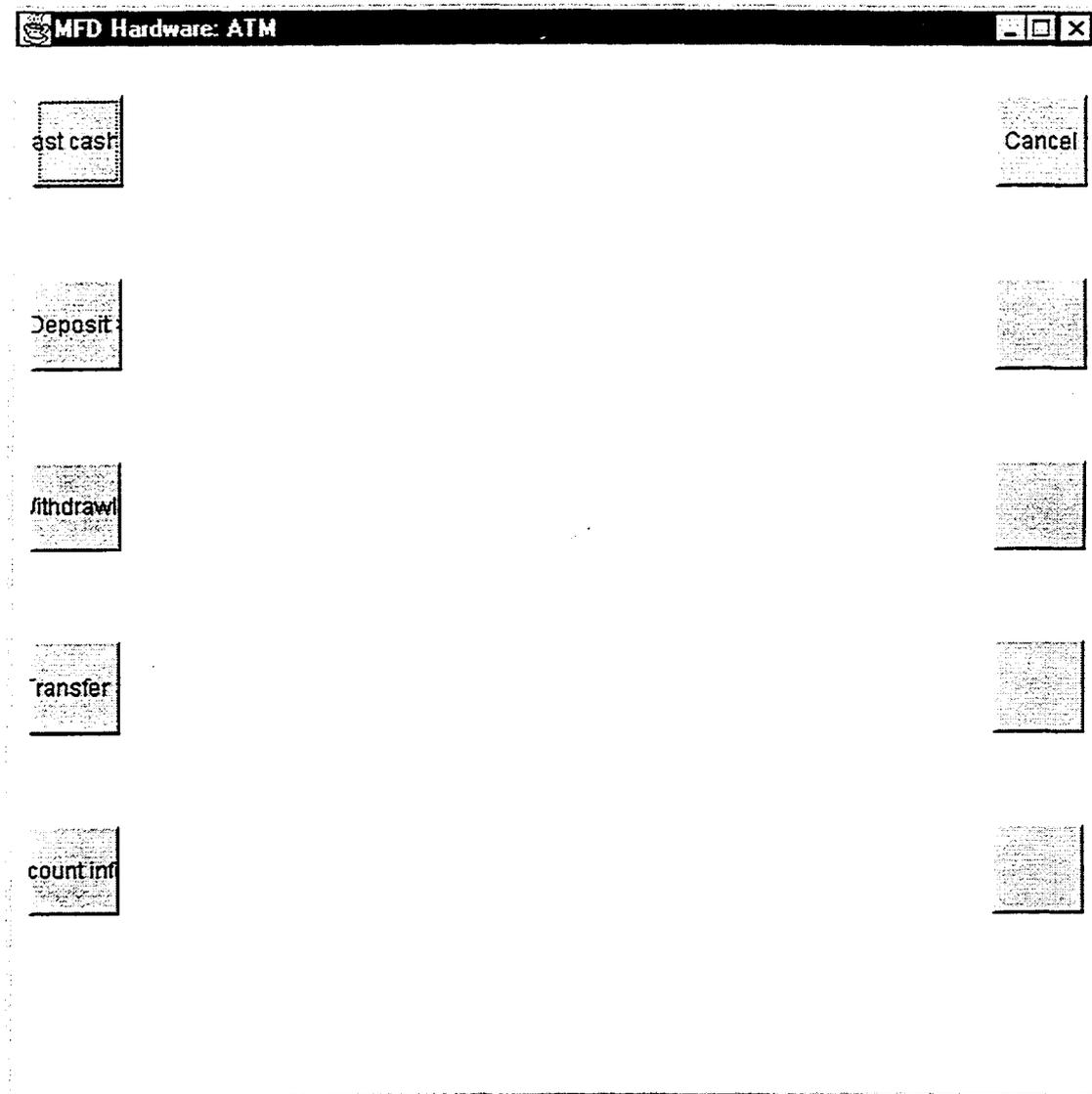


Figure 4: The MFD Hardware window after selecting the OK Password label shows the labels on the selected page.

a case, exactly which files are read likely varies from one operating system to another.

MFDTool distinguishes between three types of MFD pages. A *parent* is a page such that when that page's label is selected the MFD screen changes to reveal the contents of the page, which contains new page labels that can also be selected. A *terminator* is a page that when its label is selected the MFD screen may reveal additional information, but does not show new labels for new pages that can also be selected. A *hyperlink* is a page that when its label is selected the MFD jumps to another page someplace else in the hierarchy. Hyperlinks allow the designer to provide shortcuts between distant MFD pages. Hyperlinks also can provide a means to move up the MFD hierarchy structure, when required.

The designer specifies these types of pages in the directory file structure. Each instance of a directory in the file structure is interpreted as a parent page. For each file, MFDTool reads the first line of the file. If that first line consists solely of the word HYPERLINK, then the page is taken as a hyperlink, otherwise the page is taken as a terminator. If the designer wants to constrain a special sequence of searches that utilizes hyperlinks, the target page of the hyperlink must be identified. Details of how to do this are discussed below.²

Each page has a name associated with it (the name of its associated directory or file), and that name is the text of the label that, when selected, shows the contents of the page. For a terminator, selecting the button with its label has little effect because MFDTool is not concerned with the content shown on a page, only with the ordering of pages that might be "children" of a parent page.

The MFD information is displayed in multiple ways. First, as already noted above (Figures 3 and 4), the MFD Hierarchy window shows the MFD information, with each page name as text on its currently associated button. Second, a Hierarchy window provides a listing of the contents of the currently selected page. Figure 5 shows what this type of window looks like after loading the file ATM.hdw as above. The names associated with parent pages are given an additional '>' at their end, so the designer can quickly recognize that selecting such a page will produce movement in the hierarchy.

Movement through the hierarchy of information is done by selecting a text row. Such a selection has several effects. First, the Hierarchy window changes to show the contents of the selected page (Figure 6). Second, the MFD Hardware window also changes. The two windows are (almost) yoked so that changes to one results in changes to the other. There is an exception to this yoking for terminators and hyperlinks, as described below. Third, a status line at the bottom of the Hierarchy window shows the path of pages leading to the currently selected page. This helps the designer keep track of which page is currently selected in the hierarchy. From the Hierarchy window, the designer can move up the hierarchy by selecting the Show parent of page button, which automatically moves up to the parent of the currently selected page.

The yoke between the MFD Hardware and Hierarchy windows is not absolute. For example, selecting a terminator page in the Hierarchy window makes that page the currently selected MFD page and allows the designer to impose constraints on that page (described

²Shortcuts and hyperlinks can be created in most operating systems' file systems. However, the methods of doing this vary dramatically. To retain platform independence, MFDTool provides an internal means of specifying hyperlinks.

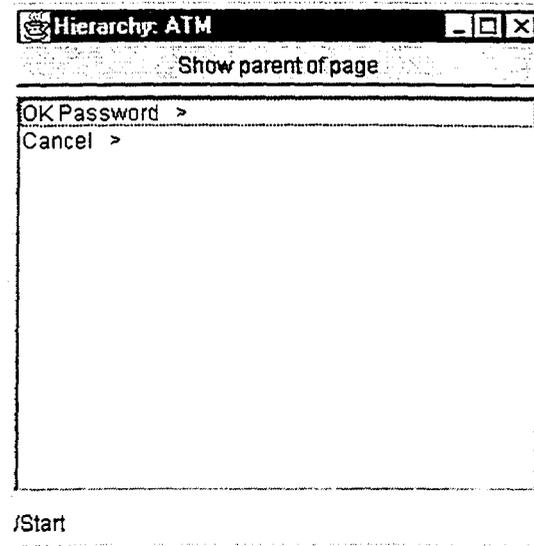


Figure 5: The MFD Hierarchy window as it appears after opening the file ATM.hdw.

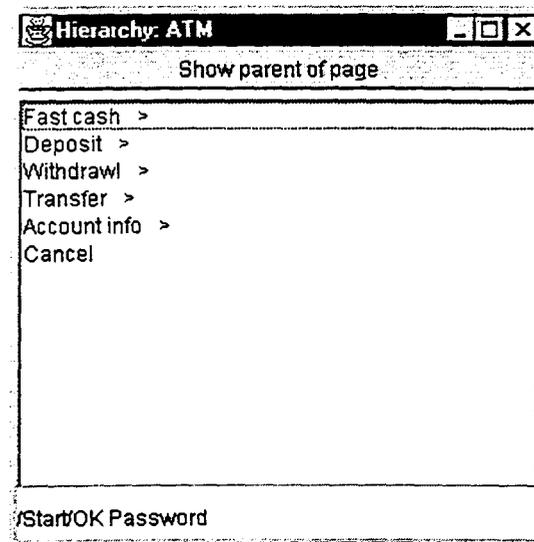


Figure 6: The Hierarchy window as it appears after selecting the page name OK Password.

below). On the MFD Hardware window, there is no noticeable change in the MFD buttons, because MFDTool does not show contents of a terminator page but instead shows the contents of the page's parent. When a terminator page is the current page, selecting a button in the MFD Hardware window will not have any effect, because the currently displayed labels are not actually children of the currently selected page. To make the MFD Hardware buttons usable, use the Show parent of page button in the Hierarchy window to make the parent of the selected terminator the new selected page. The buttons on the MFD Hardware window are now workable. This somewhat confusing relationship is necessary to allow the designer to select pages that have no noticeable effect on the MFD screen (in MFDTool anyhow), and apply constraints to those pages. For the same reason, selecting a hyperlink in the Hierarchy window does not automatically cause the MFD Hardware window to jump to the hyperlink target. Once selected in the Hierarchy window, clicking on the name of the hyperlink a second time will cause the MFD Hardware window to jump to the hyperlink target.

The selection of pages from the MFD Hardware or Hierarchy windows is also yoked to a third display of MFD page names in the main MFD window. Figure 7 shows this window after the file ATM.hdw is loaded. All forty-two pages in the ATM hierarchy are listed in the box on the left side of the MFD window. As in the Hierarchy window, a parent page has the added " >" to indicate its status. In addition, a hyperlink page has an added "-<" to indicate its status. Selecting a page from this list updates the MFD Hardware and Hierarchy windows to display that page (or that page's parent, if the selected page is a terminator). Figure 8 shows the MFD window after the OK Password page was selected in the Hierarchy window. Each page listed also has a number, $p = 0.023809524$, which indicates the default proportion of times that this page will be needed by the user. MFDTool's default is to assume that every page is needed equally often, so this number is one over the number of pages in the hierarchy, $1/42$. These proportions can be set by the designer, as described below.

2.3 Setting constraints

After the MFD hardware and hierarchy are defined, one can start to place constraints on the association of page labels to buttons. Constraints are defined in the MFD window, with the Hierarchy and MFD Hardware windows being optionally used to select various MFD pages for constraining.

The creation of a constraint begins by selecting the type of constraint that is desired. This is done with the menu choice option on the right side of the MFD window. After the type of constraint is selected, click on the New constraint button. What to do next depends on the type of constraint selected, as discussed below.

2.3.1 Global movement time

This constraint acts to minimize the average movement time as the user goes through the hierarchy to access different MFD pages. The designer has the option of providing a name for this constraint (the default is $C\#$, where $\#$ is the number of constraints currently defined). The name is listed in the text field on the bottom right of the MFD window. The designer

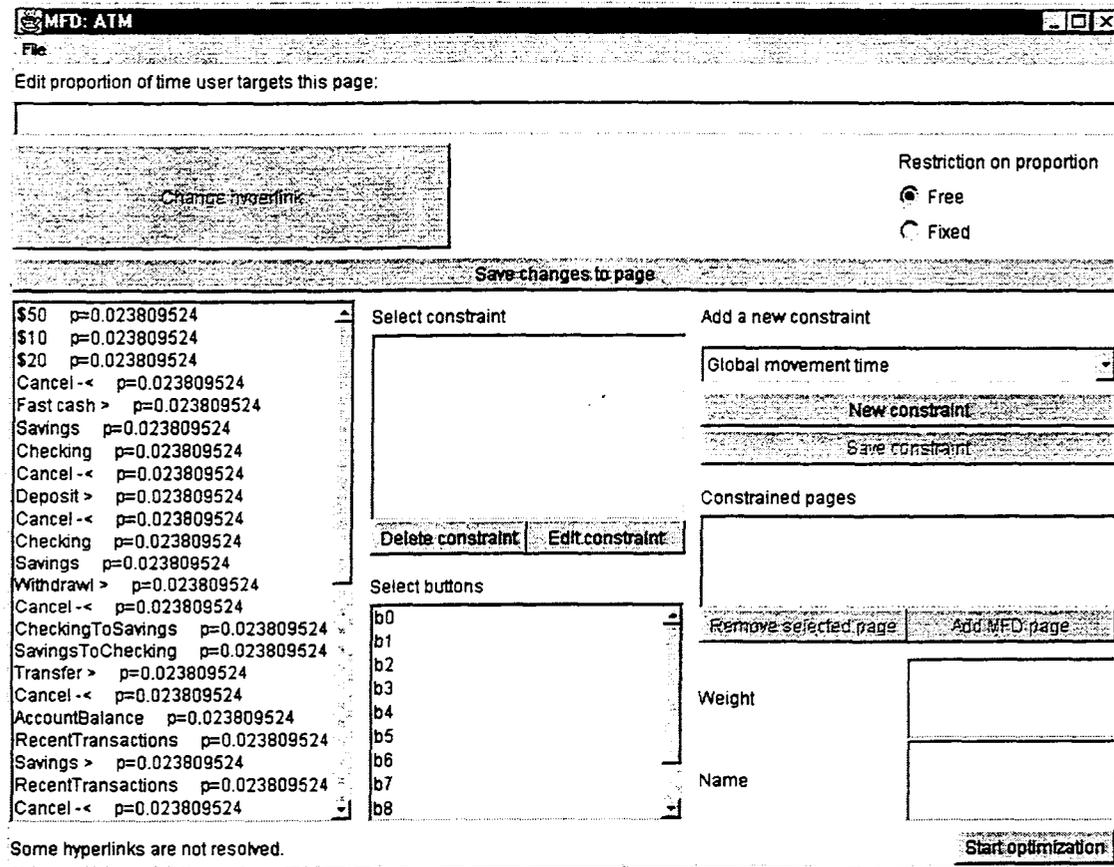


Figure 7: The MFD window as it appears after opening the file ATM.hdw.

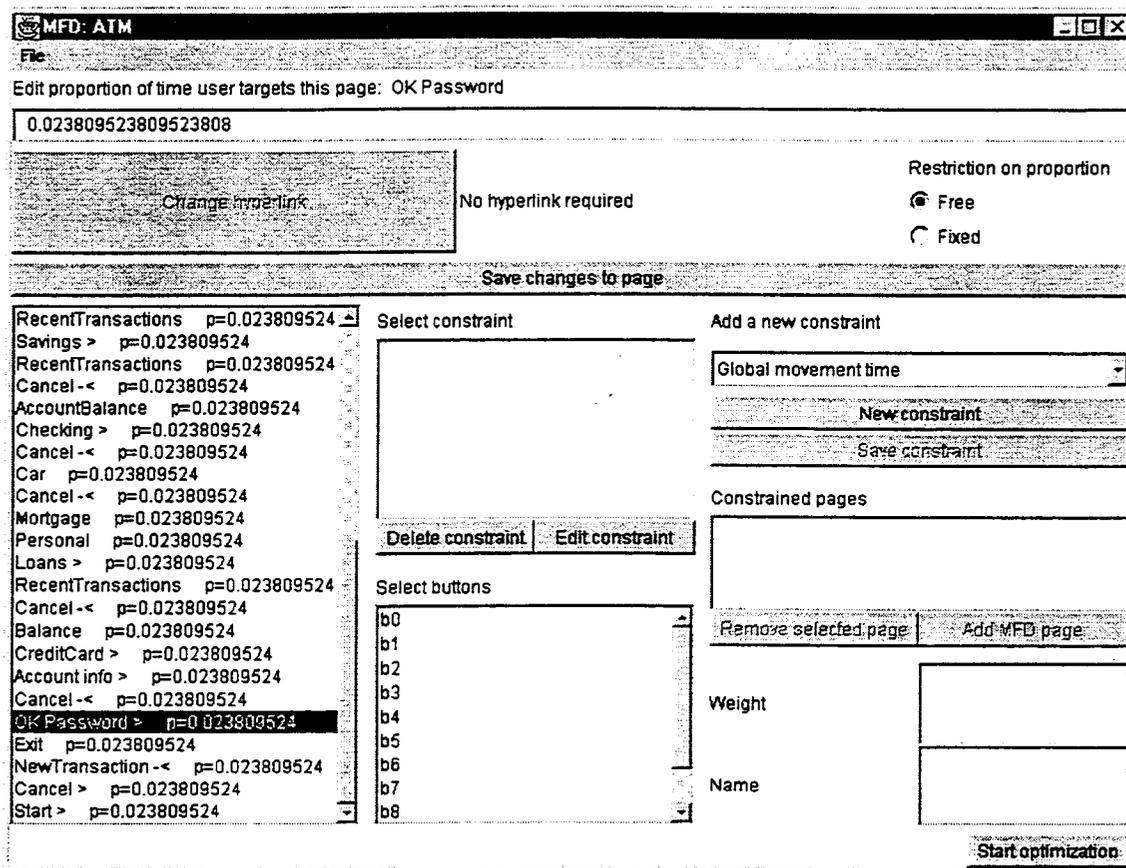


Figure 8: The MFD window as it appears after selecting the page name OK Password.

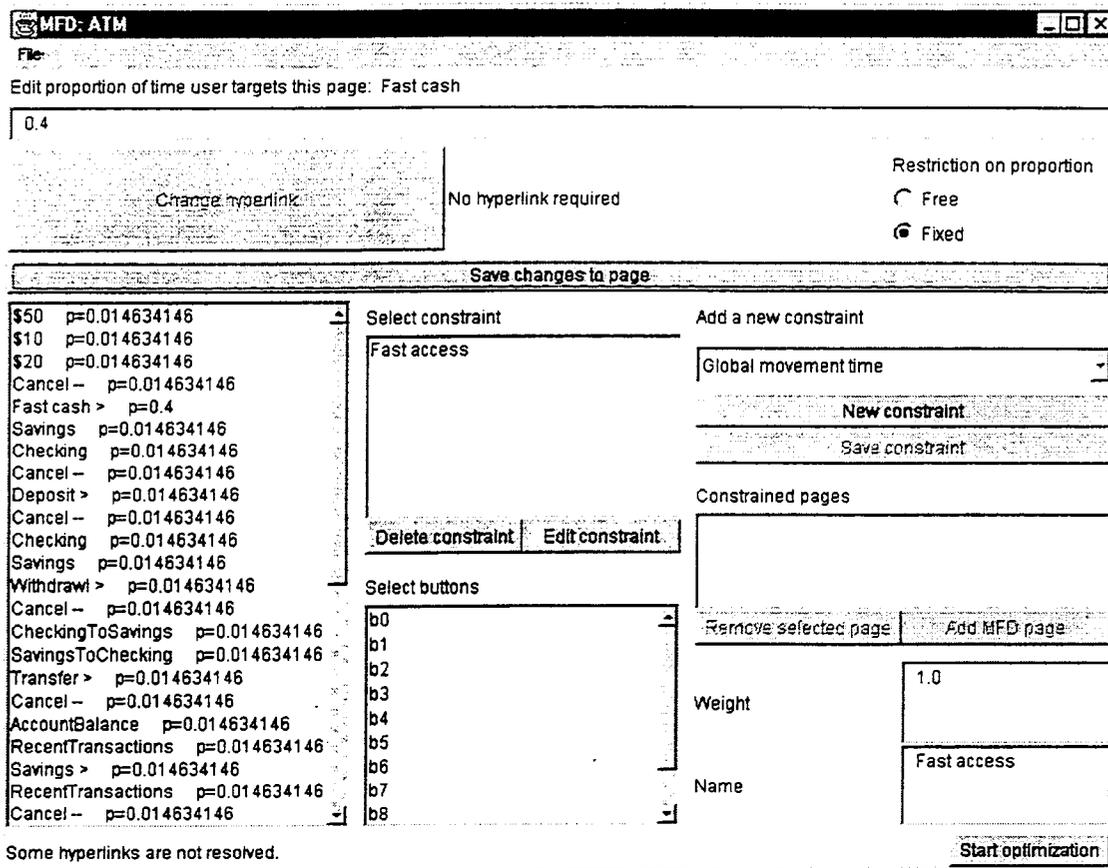


Figure 9: The MFD window as it appears after creating and saving the constraint Fast access. The upper part of the window shows the selection of information to apply a fixed proportion of 0.4 to the page Fast cash. Notice that the proportions of the other pages is smaller than in Figure 8, which reflects the renormalization after setting Fast cash proportion to 0.4.

also has the option of setting the weight for this constraint in the textfield directly above the name textfield, the default is 1.0. Once the name and weight are as desired, click on the Save constraint button. The list box in the middle of the MFD window now shows the name of the just saved constraint. Figure 9 shows what the MFD window looks like after the constraint is created and saved. There is no reason to include more than one Global movement time constraint.

From the listing of constraints, you can select (by clicking on the name) and delete or edit a constraint. For editing, clicking on the Edit constraint button loads the selected constraint's properties into the appropriate constraint fields. Changing what is in those fields and then clicking on the Save constraint button updates the properties of the constraint.

The optimization on this constraint considers the proportion of time each page is needed. The page proportions can be set at any time. Select a desired page (from any of the windows, but remember that the MFD Hardware window cannot select a hyperlink). The top part of

the MFD window shows information on the selected page. In the text field it shows the current proportion of time the page is needed by the user. To change this proportion simply edit that number. Then click on the Fixed option of the Restriction on proportion radio button. MFDTool will renormalize all the free proportions to reflect the adjustment to this page's new proportion. Making the proportion of this page fixed means that it is not subject to this renormalization.

In an ATM MFD, the Fast cash option is probably used more than any other. Figure 9 shows the MFD window just after saving the proportion for the Fast cash page to be 0.4. Clicking on the Save changes to page button updated the proportion for this page and renormalized the non-fixed proportions for other pages. Note that the proportions for the other pages are smaller than in Figure 8.

2.3.2 Pages to close buttons

This constraint has two main purposes. The first is to place specified labels on the same MFD screen as close together as possible. The second is to place specified labels on different MFD screens as close together as possible (where association with the same button is best). To create this type of constraint, select Pages to close buttons from the menu choice on the right side of the MFD window. Set the name and weight as desired. To assign selected pages to this constraint, select a page from any of the windows. It will be highlighted in the page list in the MFD window. Just above the textfield for the constraint weight is a pair of buttons and a list box. Click on the Add MFD page and the currently selected MFD page will be part of this constraint. The name of the page will appear in the list box. You can select additional pages from the MFD and add them to the constraint, or select a page from the list of constrained pages and click on the Remove selected page button to delete the page from the constraint. Once all the desired pages are added to the constraint, click on the Save constraint button, and the constraint's name will appear in the constraint list. The constraint can be deleted or edited, as described above. Figure 10 shows what the MFD window looks like after all the pages with the name Cancel are constrained to be close to each other. The weight for this constraint was set to 1000 to be absolutely certain that this constraint is satisfied, even if it means an increase in overall search time.

2.3.3 Pages to fixed buttons

This constraint allows the designer to restrict specified page labels to specified buttons. To create this type of constraint, select Pages to fixed buttons from the menu choice on the right side of the MFD window. Set the name and weight of the constraint as desired. Assign selected pages as for the Pages to close buttons constraint, above. In addition, from the list of Select buttons, click on the names associated with the buttons you want the pages to be restricted to.³

³At the moment the interface here is less than optimal. The buttons are named b_0, \dots, b_n , where there are n buttons. The order of the buttons is as given in the MFD hardware file, which in the ATM example is ATM.hdw. Future versions of MFDTool will offer an improved system of identifying which name in the button list corresponds to which button in the MFD Hardware window.

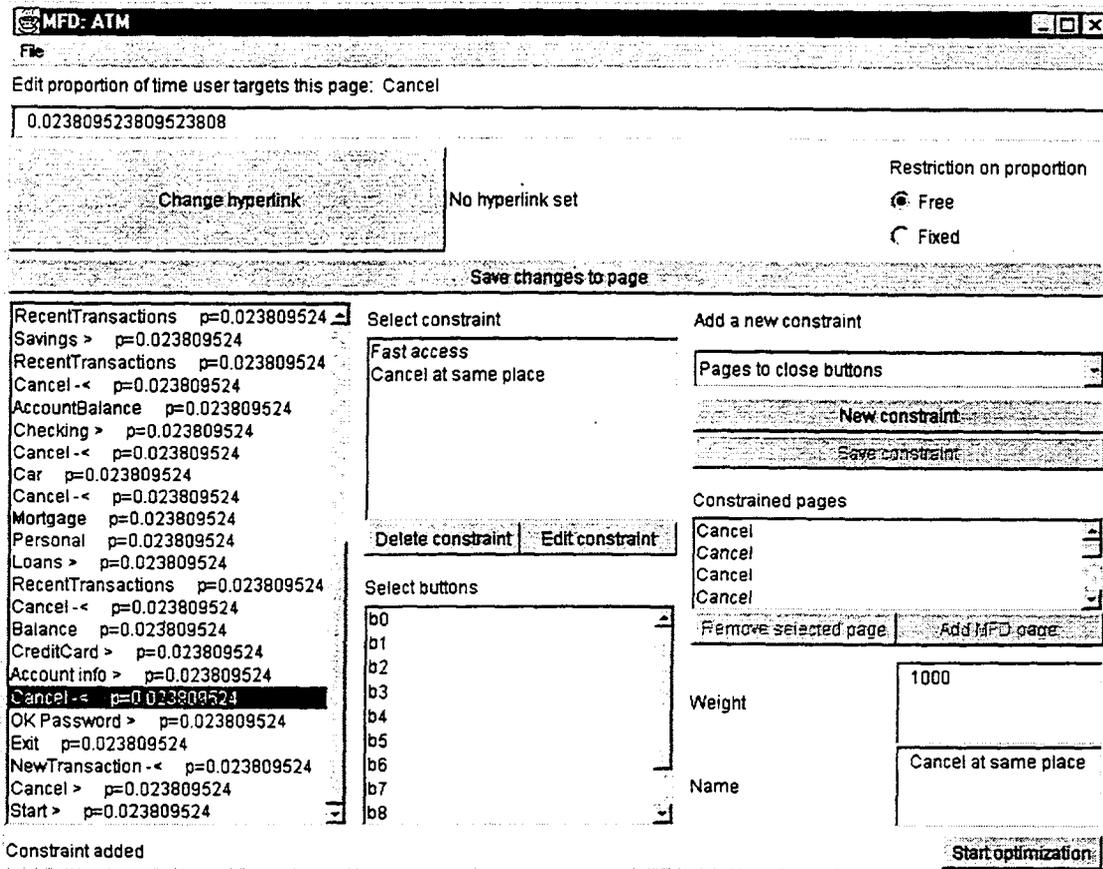


Figure 10: The MFD window as it appears after creating and saving the constraint Cancel at same place. The list Constrained pages shows all the pages restricted by this constraint.

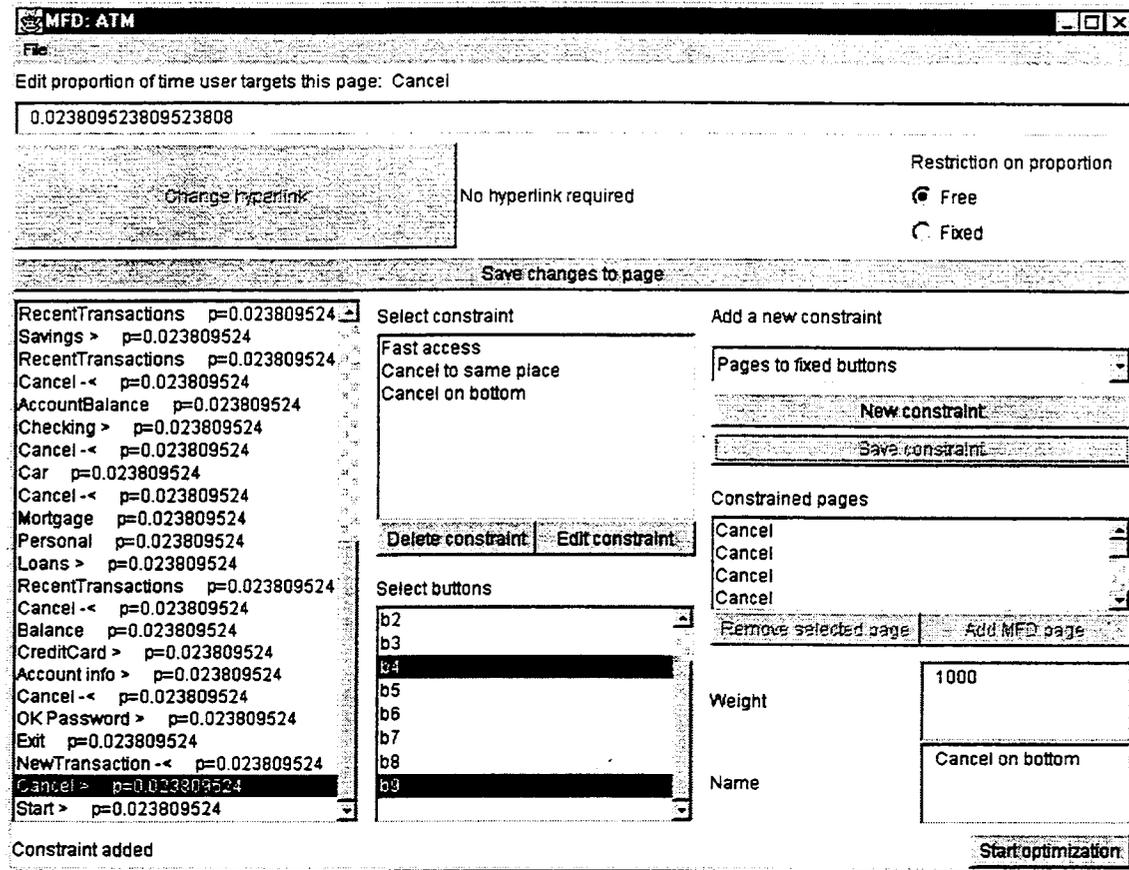


Figure 11: The MFD window as it appears after creating and saving the constraint Cancel on bottom. The list Constrained pages shows all the pages restricted by this constraint. The list Select buttons shows the buttons that the constrained pages must be restricted to.

Figure 11 shows the MFD window after the designer has introduced a constraint to place the Cancel labels to be on either of the two bottom buttons. Notice that in combination with the constraint to place all the Cancel labels on the same button, the optimized MFD design should put all the Cancel labels on either the bottom right or the bottom left button.

2.3.4 Path movement time

This constraint applies to a designer-defined special sequence. This type of constraint is most applicable to a MFD that includes hyperlinks. Before discussing how to apply this constraint, the following describes how to define hyperlinks.

A page that consists of a hyperlink is a shortcut to a different part of the MFD hierarchy. Selecting a properly defined hyperlink in the MFD Hardware window makes the hyperlink target page the currently selected page and displays that page (or its parent if the target is a terminator). When the directory structure that defines the MFD hierarchy is loaded into

MFDTool, there is no way to identify the targets of hyperlink pages. The designer must define the hyperlinks within MFDTool. This is done from the MFD window.

Hyperlinks that have not had their target defined are shown in the page listing on the MFD window with an additional ‘‘-<’’ added to their name. If a hyperlink is selected from this list (or from the Hierarchy window), the top part of the MFD window displays information on the page, its proportion of being needed, whether that proportion is fixed or not, and the hyperlink target. The button labeled **Change hyperlink** is also clickable when the selected page is a hyperlink. Clicking on this button puts MFDTool into a mode whereby selecting a page from any window places that page as a potential target for the hyperlink. This is visible in the text to the right of the **Change hyperlink** button, which displays the selected page name. When the desired page is set as the target for the hyperlink, click on the **Save changes to page** button and the hyperlink has now been defined. Once a hyperlink is defined, the page listing on the MFD window replaces the ‘‘-<’’ with ‘‘--’’. Figure 12 shows the MFD window right after one of the **Cancel** pages has had its hyperlink set to the **Cancel** page at the top of the hierarchy (this **Cancel** page is not a hyperlink, but a parent page that shows options for the user after the **Cancel** button has been pressed). If a hyperlink is not defined, MFDTool treats it as a terminator page.

To create a path movement time constraint, select **Path movement time** from the menu choice on the right side of the MFD window. Set the name and weight of the constraint as desired. The designer now needs to assign pages to the constraint *in the same order* that the user will travel through the pages along the “special path.” Select the first page along the path, and click on the **Add MFD Page** button. For each subsequent page along the path, select that page and click on the **Add MFD Page** button. The order that the pages are added to the constraint is the order that MFDTool will assume the user will move through the hierarchy.

Once the path is defined, click on the **Save constraint** button. Figure 12 shows the MFD window after saving a short **Path movement time** constraint.

2.4 Optimizing

Once all the desired constraints are set, click on the **Start optimization** button on the lower right of the MFD window. Depending on your computer, the size of the hierarchy, and the complexity of the constraints, the optimization process may take a few tens of minutes to several hours.

There is no way to adequately describe the effect of optimization in written text. A previously optimized design is included with MFDTool. Start MFDTool and go to **File**→**Open**. From the open file window, select the directory **ATM** and then the file **ATMbest.mfd**. A new set of **MFD**, **MFD Hardware**, and **Hierarchy** windows will appear. Clicking through the **MFD Hardware** and **Hierarchy** selections will give a good idea of the assignment of page labels to buttons.

The optimization is, of course, relative to the defined constraints. You can explore the chosen constraints from the MFD window. Of special note, this example set the search probabilities of two pages **OK Password**→**Fast cash** and **OK Password**→**Withdraw**→**Checking** to

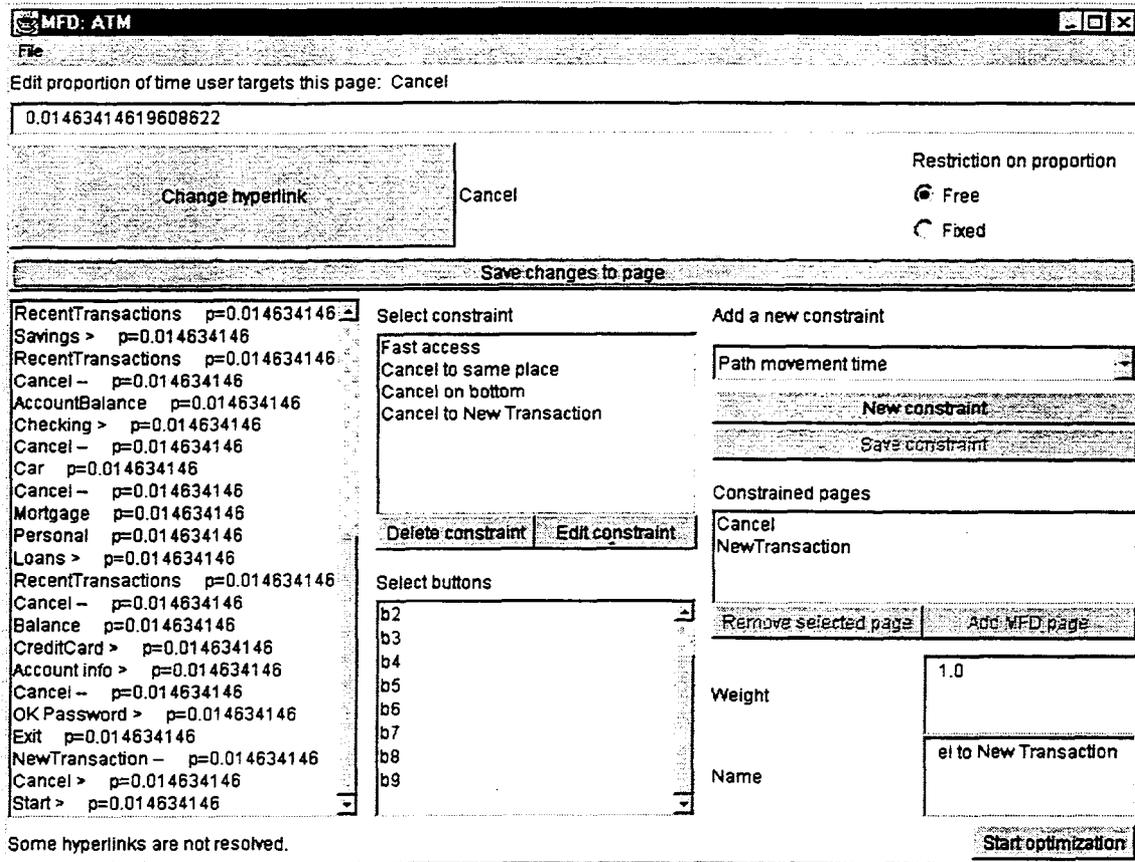


Figure 12: The MFD window as it appears after creating and saving the constraint Cancel to New Transaction. The list Constrained pages shows the path of pages restricted by this constraint. In this case, the path starts at the Cancel hyperlink under Fast cash and goes to the New Transaction page under the Cancel page near the top of the hierarchy. The top part of the window shows that hyperlink Cancel, under the Fast cash page, is a hyperlink to the Cancel page at the top of the hierarchy. Thus, selecting Cancel under Fast cash jumps the MFD to the Cancel screen. From that screen, NewTransaction can be selected. MFDTool will assign page labels to buttons so that movement time is minimized along this path of pages.

0.4. The resulting design minimizes the movement between buttons to reach these pages.

Note too that the resulting design deals with a number of subtle issues. For example, from the OK Password page note that Account info is on the middle button on the left side. After clicking on the OK Password button (top, left side), one will have to move down to click on this button. One must move down yet two more buttons to reach the Transfer page. The Account info button is closer to the OK Password button than the Transfer button. This is because the Account info page has more information underneath it than the Transfer page. This means that, with the assumption that every "free" page is needed equally often, the user will go through the Account info page more often than the Transfer page. Thus, the best design is to assign the Account info page label to a button that is closer to the OK Password button.

Also note that after selecting a button, the page labels on the next screen tend to be right around the button that was just clicked. This minimizes movement. Other constraints are also satisfied. All the Cancel labels are on the same button (bottom, right). The dollar amounts under the Fast cash page are lined up in order.

In short, once the MFD hardware, hierarchy, and constraints are defined, the optimization is handled by the computer. The optimization is exceedingly thorough, and considers many details that a human designer will likely not have the time to deal with.

2.5 Saving MFDs

Once a MFD and its properties are defined, the entire set can be saved as a file. Simple select File→Save. In the file window, select a directory and type a filename (it is recommended that you use the extension .mfd, though this is not absolutely necessary). This file holds all the properties of the MFD and can be opened later.

3 Conclusions

MFDTool provides a means of optimizing the association of MFD hierarchical information with MFD buttons. Creating an effective association is difficult because of the large number of variables involved in the task. MFDTool quantifies the variables, thereby allowing standard optimization approaches to be applied to the problem.

MFDTool should rarely be used in isolation from the rest of the design process. Instead, it will probably be most beneficial as a collaborator with a designer. That is, a designer may create one hierarchical organization and hardware configuration and then run MFDTool to find the optimal layout of information. From that starting point, the designer can consider the effect of changing the hierarchical organization or hardware configuration, and re-running MFDTool for each change. The only fair comparison across different hierarchical or hardware configurations is relative to their optimal association. Such comparisons would have been nearly impossible in the past because the association itself was very difficult to optimize. MFDTool greatly simplifies this process, thereby allowing the designer to compare a larger set of possible designs.