University of Bielefeld WS 2004/2005

Faculty of Technology

Networks and Distributed Systems

Diploma Thesis

Formal Task Analysis of Graphical System Engineering Software Use

Thilo Paul-Stüve

10 March 2005

Examiners: Prof. Peter B. Ladkin, Ph.D.
Dipl. Inform. Bernd Sieker

Declaration

I completed this work without assistance and used no other sources or aids than those listed. All quotations are marked as such.

Bielefeld, 10 March 2005 Thilo Paul-Stüve

Contents

List of Figures

1. Introduction

Much good design evolves: the design is tested, problem areas are discovered and modified, and then it is continually retested and re-modified until time, energy, and resources run out. [. . .] Over time, this process results in functional, aesthetically pleasing objects.

Don Norman [Norman 88]

This thesis describes the analysis of the use of a graphical software to support an analyst in applying the Why-Because Graph method, which is an essential part of the Why-Because Analysis method. The aim is to improve the user interface of existing software to better support the user in applying the Why-Because Graph method.

The tasks that specify the Why-Because Analysis are determined using the Hierarchical Task Analysis method. Then the use of the existing software, YB-Edit, to accomplish specific tasks during a Why-Because Analysis, is examined using GOMSL, a GOMS method variant. A new interaction design based on the results of the analysis is introduced. It is analysed with GOMSL as well and the outcomes of both analyses are finally compared.

1.1. Overview

1.1.1. Why-Because Analysis

The Why-Because Analysis (WBA) is a failure analysis method designed for complex systems consisting of components of different types, that are highly influenced by their environment, such as airplanes or cars.

WBA consists of the Why-Because Graph (WB Graph or WBG) method and the verification of its results by formal proof. The result of the WBA is a consistent description of the failure of such a system. Please see [Loer 98, Ladkin 01] for a detailed description of the WBA method. An introduction can be found in [Ladkin 99].

1.1.1.1. The Why-Because Graph Method

The WB Graph method is based on David Lewis's formal semantics for causality [Ladkin 01]. The basic idea is that an event or state description A is a necessary causal factor of an event or state description B only if wether B occurs or not depends on wether A occurs or not.

The WB Graph method is applied by collecting all events and states that are important to the course of failure. These are called a facts. Then all causal relations between them are determined.

A Why-Because Graph states all causal relations between the facts that led to the failure of a system. The nodes of the WB Graph represent the facts, and the edges the causal relations between them. The system failure (mishap) is represented by the top node of the WB Graph.

The possibility to gain easy access to the course of events by looking at a graphical representation of the WB Graph helps to communicate the results of a WBA.

1.1.2. YB-Edit

YB-Edit, described in [Sieker 04], is a graph drawing tool with a graphical user interface specialised in supporting the Why-Because Graph method described in [Loer 98, Ladkin 99, Ladkin 01]. It evolved from CI-Edit, which was developed by Joachim Weidner, and CI-Edit2 developed by Bernd Sieker.

The CI-Edit programs were designed to handle Causal Influence Diagrams, which are part of the Causal System Analysis, a method to analyse a systems possible failures, described in detail in [Ladkin 01].

The purpose of YB-Edit is to support the analyst in applying the WB Graph method in a more intuitive way than entering the corresponding Why-Because Script, a language for textual description of WB Graphs defined in [Loer 98, Ladkin 01], in a text editor. It enables the analyst to create the nodes and edges of the WB Graph by using the mouse to manipulate a visual representation of the graph. The changes made to the graph are displayed immediately. YB-Edit further utilises the "verteiltes Datenarchivierungsystem" (VDAS), described in [Hennig 03], which allows safe archiving of data and cooperative use of shared graphs. The WB Graphs created with YB-Edit can be exported to PostScript page description language files for print or presentation.

YB-Edit is the most popular software tool especially designed for the creation of WB Graphs.

1.1.3. The "Friendly Fire" Accident

The "Friendly Fire — Deaths Traced to Dead Battery" case is used to teach performing WBA in seminars. Three U.S. Soldiers were accidentally killed because they changed the batteries of their GPS receiver. Before the batteries of the device died, they targeted a Taliban Outpost to call in an airstrike for its coordinates.

They did not take into account that the loss of electric power caused the receiver to initialise itself with its own coordinates and therefore called in the airstrike for their own coordinates.

The corresponding List of Facts and WB Graph are shown in Figure 1.1, and Figure 1.2.

```
List of Facts for 'Friendly Fire' Deaths Traced to Dead Battery
Taliban Targeted but U.S. Forces Killed

 1.     3 SF-Soldiers killed
 2.     20 SF-Soldiers wounded
 3.     2K-pound satellite guidd bomb hit SF-Soldiers location
 4.     SF-Soldiers aimed for Taliban outpost and aquired GPS coords
 5.     SF-Soldiers called in first airstrike
 6.     F/A-18 responded to first call for airstrike
 7.     F/A-18 successfully hit Taliban Outpost
 8.     Battery of GPS device died
 9.     SF-Soldiers change battery
 10.    GPS receiver initialises itself with own position
 11.    Taliban Outpost position differs from own position only in seconds
 12.    SF-Soldiers were not able to tell the difference between Taliban
Position and init-position
 13.    SF-Soldiers were not aware on GPS reinitialisation procedures
 14.    SF-Soldiers believed coords in GPS to be the Taliban Outpost coords
 15.    SF-Soldiers called in second airstrike
 16.    B-52 responded to second call for airstrike
 17.    B-52 targeted SF-Soldiers position

a. B-52 released satellite guided bomb
b. SF-Soldiers wanted to attack Taliban Outpost
```

Figure 1.1.: "Friendly Fire" List of Facts.

1.1.4. Human-Computer Interface

> The human-computer interface is easy to find in a goss way — just follow a data path outward from the computer's central processor until you stumble across a human being. [Card 83]

The human computer interface is the way how tasks are accomplished with a computer system. It consists of the parts of the computer system which can be seen, heard, or felt by the user, and the operations of the user to control its operation. It is "what you do and how it responds" [Raskin 00]. Some terms related to user interface design are briefly introduced here. For detailed introductions see [Dix 98, Raskin 00].

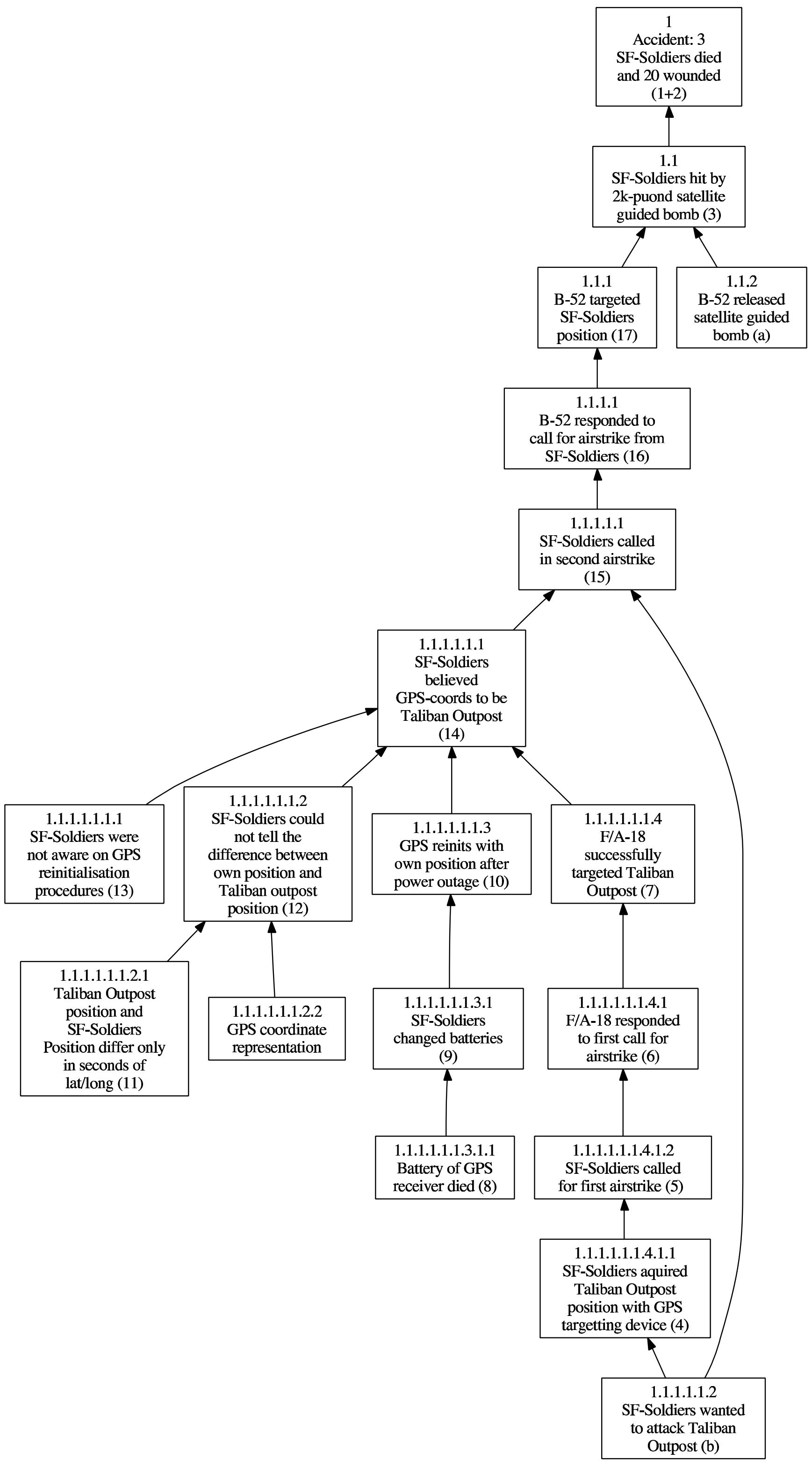

1
Accident: 3 SF-Soldiers died and 20 wounded (1+2)
1.1
SF-Soldiers hit by 2k-puond satellite guided bomb (3)
1.1.1
B-52 targeted SF-Soldiers position (17)
1.1.2
B-52 released satellite guided bomb (a)
1.1.1.1
B-52 responded to call for airstrike from SF-Soldiers (16)
1.1.1.1.1
SF-Soldiers called in second airstrike (15)
1.1.1.1.1.1
SF-Soldiers believed GPS-coords to be Taliban Outpost (14)
1.1.1.1.1.1.1
SF-Soldiers were not aware on GPS reinitialisation procedures (13)
1.1.1.1.1.1.2
SF-Soldiers could not tell the difference between own position and Taliban outpost position (12)
1.1.1.1.1.1.3
GPS reinits with own position after power outage (10)
1.1.1.1.1.1.4
F/A-18 successfully targeted Taliban Outpost (7)
1.1.1.1.1.1.2.1
Taliban Outpost position and SF-Soldiers Position differ only in seconds of lat/long (11)
1.1.1.1.1.1.2.2
GPS coordinate representation
1.1.1.1.1.1.3.1
SF-Soldiers changed batteries (9)
1.1.1.1.1.1.4.1
F/A-18 responded to first call for airstrike (6)
1.1.1.1.1.1.3.1.1
Battery of GPS receiver died (8)
1.1.1.1.1.1.4.1.2
SF-Soldiers called for first airstrike (5)
1.1.1.1.1.1.4.1.1
SF-Soldiers aquired Taliban Outpost position with GPS targetting device (4)
1.1.1.1.1.2
SF-Soldiers wanted to attack Taliban Outpost (b)

Figure 1.2.: "Friendly Fire" WB Graph as developed under my observance.

1.1.4.1. Direct Manipulation Interface

With a direct manipulation interface, the user manipulates a graphic representation of the underlying data. [Shneiderman 82] introduces the term direct manipulation interface and states the principles of direct manipulation:

Principles of Direct Manipulation

- the continuous representation of the object of interest
- gestures (physical actions) or labelled buttons presses instead of complex syntax
- rapid incremental reversible operations whose impact on the object of interest is immediately visible

1.1.4.2. Direct Engagement versus Indirect Engagement

Interfaces can be classified in interfaces with direct engagement of the user, where the user feels as direct actor, and interfaces with indirect engagement, where the user interacts through an invisible intermediary who executes commands [Frohlich 97].

1.1.4.3. Usability

The usability of an interface is sometimes described as the "ease of use". A more exhaustive definition can be found in [Macaulay 95]: Usability is the ease of learning, the ease of use, the flexibility of use, the effectiveness of use, and the user satisfaction with the system.

1.1.4.4. Fitt's Law

Fitt's Law defines the time to reach a screen object with a pointing device. The movement time to position the pointer over the target object is dependent on the distance of the actual pointer position to the target object, and the size of the target object. A common form is: $movement\ time = a + b\,\log_2(distance/size + 1)$, where a and b are constants that are empirically determined. See [Dix 98], or [Raskin 00].

1.1.4.5. Locus of Attention

The locus of attention is the one location of sensory input attended to at a given time. While the focus of attention implies volition, the locus of attention can not be controlled completely. A good user interface avoids the necessity to shift the locus of attention to control the program, because this distracts the user from the task he wishes to complete. See [Raskin 00].

1.1.4.6. Modes

A human-computer interface is modal, when the system provides several different system state dependent responses to one user action, while the actual state of the interface is not the user's locus of attention. Different modes in human-computer interfaces are often irritating to the user and can lead to handling errors, which might cause damage to work. Therefore they should be omitted. See [Raskin 00] or [Norman 88].

1.1.5. Task Analysis

Task analysis are methods used to describe and evaluate human-machine or human-human interactions. They help analyse, what tasks must be carried out to achieve a specific goal.

Different task analysis methods provide the analyst with practices and techniques to collect and organise information, and to make judgements on basis of the obtained information. A detailed overview of task analysis methods in regard to safety, productivity issues, and availability standards is given in [Kirwan 92]. Task analysis procedures of use in the field of learning and training systems are introduced in [Jonassen 89]. [Hackos 98] describes the use of task analysis for interface design.

A specific task analysis method that gives a detailed view of how a user interacts with a system is the GOMS method, which is described later.

1.2. Hierarchical Task Analysis

Hierarchical Task Analysis (HTA) was introduced by John Annett and Keith Duncan in 1967 [Annett 67]. It was developed for task data collection and organisation for use in training applications and is now a general method for task representation applied in several contexts. A task, as described in [Kirwan 92], is the method to attain a goal. It is constrained by availability and cost of materials , provided equipment and facilities, availability and cost of services, time obligations, legal obligations, and preferences of management and staff.

The result of a HTA is a hierarchy of operations and plans that must be carried out to get a task done. The level of detail of the analysis is determined by the analyst. It arises from the motive of the analysis and the kind of the analysed task.

1.2.1. Application

HTA can be used through the whole design process of a system, or on an existing system to determine how a task should or is carried out. It is used for interface design, work organisation, user manuals, training, and error analysis.

1.2.2. The Procedure

Information is obtained through interviews or observation of different sources, preferable experts of the field examined. The HTA method is to be understood as a data collection method. It structures the process of data collection. Before starting, the purpose and level of detail of the analysis must be determined.

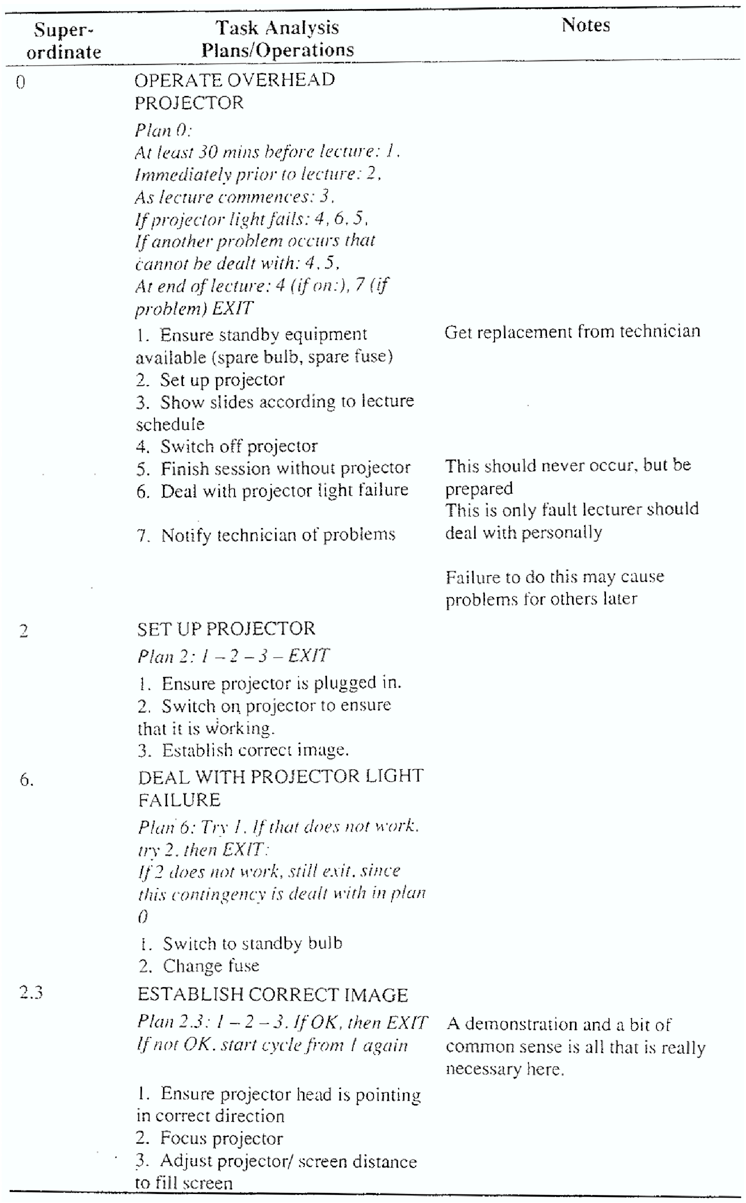
First, the goal of the task has to be determined, where goal is the desired state of the system under control. Next, the goal has to be described as a set of sub-operations that lead to it, and plans when to carry them out. The sub-operations are described iteratively as set of sub-operations until the desired level of detail is reached.

Operations are units of behaviour: something people do. The ability to select appropriate actions after perceiving information, to carry them out, and to perceive feedback is implied. Plans specify under which circumstances what operation is carried out. A plan can be a sequence of actions, or conditional actions dependent on time, process condition, or instructions like loops or options.

Experience in carrying out a HTA is of great advantage, since there are no strict rules of how to decompose tasks in the correct way.

1.2.3. Representation

The information collected is usually arranged in tables (Figure 1.3) and in hierarchical diagrams (Figure 1.4). In tables detailed design notes can be added easily, so they are often more thorough than the corresponding hierarchical diagrams. They can be used to record and to communicate the analysis. Hierarchical diagrams are easier to assimilate and give a very good overview of the structure. They are often used for presentation of the results of an analysis.



Super-ordinate	Task Analysis Plans/Operations	Notes
0	OPERATE OVERHEAD PROJECTOR	
	Plan 0: At least 30 mins before lecture: 1. Immediately prior to lecture: 2. As lecture commences: 3. If projector light fails: 4, 6, 5. If another problem occurs that cannot be dealt with: 4, 5. At end of lecture: 4 (if on:), 7 (if problem) EXIT	
	1. Ensure standby equipment available (spare bulb, spare fuse)	Get replacement from technician
	2. Set up projector	
	3. Show slides according to lecture schedule	
	4. Switch off projector	
	5. Finish session without projector	This should never occur, but be prepared
	6. Deal with projector light failure	This is only fault lecturer should deal with personally
	7. Notify technician of problems	Failure to do this may cause problems for others later
2	SET UP PROJECTOR	
	Plan 2: 1 – 2 – 3 – EXIT	
	1. Ensure projector is plugged in.	
	2. Switch on projector to ensure that it is working.	
	3. Establish correct image.	
6.	DEAL WITH PROJECTOR LIGHT FAILURE	
	Plan 6: Try 1. If that does not work, try 2, then EXIT: If 2 does not work, still exit, since this contingency is dealt with in plan 0	
	1. Switch to standby bulb	
	2. Change fuse	
2.3	ESTABLISH CORRECT IMAGE	
	Plan 2.3: 1 – 2 – 3. If OK, then EXIT If not OK, start cycle from 1 again	A demonstration and a bit of common sense is all that is really necessary here.
	1. Ensure projector head is pointing in correct direction	
	2. Focus projector	
	3. Adjust projector/ screen distance to fill screen	

Figure 1.3.: An example of a HTA table: An analysis of overhead projector use [Kirwan 92].

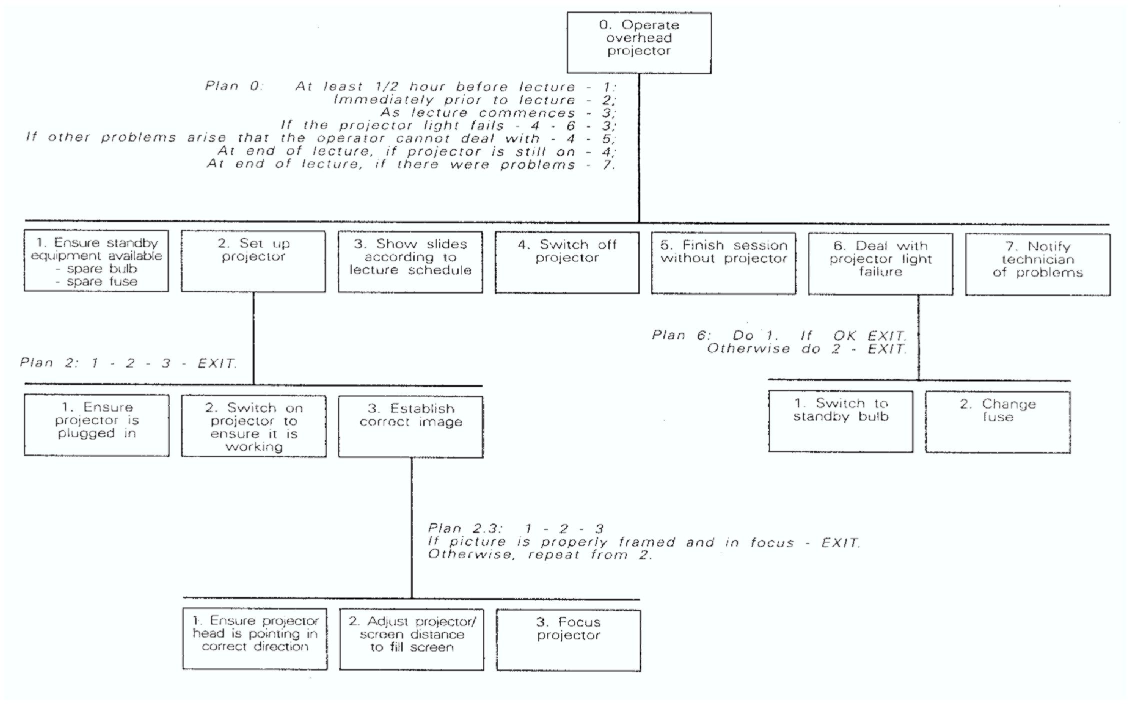

0. Operate overhead projector
Plan 0: At least 1/2 hour before lecture - 1;
Immediately prior to lecture - 2;
As lecture commences - 3;
If the projector light fails - 4 - 6 - 3;
If other problems arise that the operator cannot deal with - 4 - 5;
At end of lecture, if projector is still on - 4;
At end of lecture, if there were problems - 7.
1. Ensure standby equipment available - spare bulb - spare fuse
2. Set up projector
3. Show slides according to lecture schedule
4. Switch off projector
5. Finish session without projector
6. Deal with projector light failure
7. Notify technician of problems
Plan 6: Do 1. If OK EXIT. Otherwise do 2 - EXIT.
1. Switch to standby bulb
2. Change fuse
Plan 2: 1 - 2 - 3 - EXIT.
1. Ensure projector is plugged in
2. Switch on projector to ensure it is working
3. Establish correct image
Plan 2.3: 1 - 2 - 3
If picture is properly framed and in focus - EXIT.
Otherwise, repeat from 2.
1. Ensure projector head is pointing in correct direction
2. Adjust projector/screen distance to fill screen
3. Focus projector

Figure 1.4.: The example shown in Figure 1.3 in diagram form [Kirwan 92].

1.3. The Goals, Operators, Methods, and Selectors Family

1.3.1. GOMS

The Goals, Operators, Methods, and Selectors method (GOMS) was introduced by Stuart K. Card, Thomas P. Moran, and Allen Newell for analysis of text processor use in 1983 [Card 83].

GOMS is a method to produce quantitative and qualitative predictions of single expert user interaction with passive systems, that react on actions of the user, and active systems, that are able to produce actions by themselves (e.g. a computer game) [John 95a]. It is a serial stage architecture analysis with program like structures. A GOMS analysis goes down to a level of detail that is that of basic tasks like pressing a mouse button, moving the mouse to a specific object on the screen, or pressing a key on the keyboard. Basic tasks have a specific execution time, that has been researched in experiments. To make time predictions the specific execution times of the operations required to achieve a goal are added up.

The complexity of a method can be measured by the complexity of the modelling process and the straightness of its structure.

1.3.1.1. The Model Human Processor

The Model Human Processor (MHP) was introduced by Stuart K. Card, Thomas P. Moran, and Allen Newell in 1983 [Card 83]. It is a model of human information processing with parallel stages.

Human information processing is modelled by processors and storage systems: Perceptual processors acquire, reorganise, and store information in working memory. A cognitive processor works on the information stored in working memory and commands the motor processors, which carry out physical actions. All processors operate serially on their own and in parallel with each other, while each processor has a specific cycling time.

The MHP forms the theoretical framework for GOMS.

1.3.1.2. GOMS Basic Definitions

[John 96, Dix 98] further explain the GOMS components:

Goals A goal is a structure that defines the state of affairs to be achieved and determines the possible methods to accomplish. It further represents a memory point to return to upon error to determine what is desired, what methods are available, and what has been tried. Goals are normally broken down into subgoals, which must be accomplished to achieve a higher level goal.

Operators An operator is the elementary perceptual, motor or cognitive act that changes the user's mental state or affects the task environment. The execution time of an operator is approximated by a constant, a probability distribution, or function of a parameter. In contrast to the MHP, GOMS operators are not executed in concurrence.

Methods A method describes the procedure for accomplishing a goal and is made up of a conditional sequence of subgoals and operators with test on the user's contents of working memory and on the status of the task environment. They represent skills rather than plans.

Selectors Selection rules choose the appropriate methods to achieve a goal on base of the task environment features or decisions by the user. They are rules like "if this and that is true then use method XY".

1.3.1.3. Application

GOMS can be used through the whole design process of a system, or on an existing system to make predictions about overall performance time and time to perform single tasks. Evaluation of alternative existing systems and rival system designs are possible even at specification stage. GOMS is used for interface design, user manuals, error analysis, and human cost calculation.

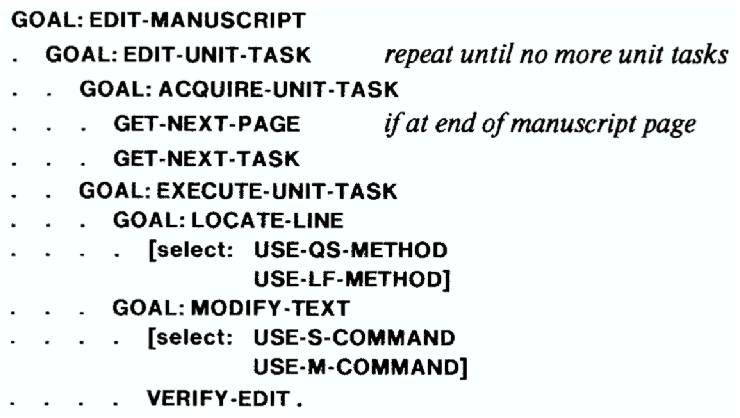
1.3.1.4. Performing a GOMS Analysis

The information necessary to build a GOMS model is obtained by performing skills and knowledge acquisition techniques like questionnaires, structured interviews, verbal protocols, Task Analysis for Knowledge Description (TAKD, see [Dix 98] for an introduction), or HTA (see section 1.2).

When sufficient information is collected, the main goal has to be determined first, followed by the method to accomplish it. The method has to be made up of higher level subgoals at this point of the analysis. If necessary, selection rules have to be specified for the method.

For every sub goal, a method with its own selection rules has to be determined. The deeper the level of analysis, the more methods consist of operators than of sub goals.

The process has to be continued recursively until all lowest level methods are specified only by operators. Doing so brakes the main goal down completely to basic acts of human performance. An example is shown in Figure 1.5.



```
GOAL: EDIT-MANUSCRIPT
.   GOAL: EDIT-UNIT-TASK          repeat until no more unit tasks
.   .   GOAL: ACQUIRE-UNIT-TASK
.   .   .   GET-NEXT-PAGE         if at end of manuscript page
.   .   .   GET-NEXT-TASK
.   .   GOAL: EXECUTE-UNIT-TASK
.   .   .   GOAL: LOCATE-LINE
.   .   .   .   [select: USE-QS-METHOD
                         USE-LF-METHOD]
.   .   .   GOAL: MODIFY-TEXT
.   .   .   .   [select: USE-S-COMMAND
                         USE-M-COMMAND]
.   .   .   .   VERIFY-EDIT.
```

Figure 1.5.: GOMS example taken from [Card 83].

There are no detailed rules of how to decompose the achievement of a goal in a useful way, so experience is of great use. Using structuring information acquisition techniques such as HTA further helps in performing a GOMS analysis, since the embodied organisation can be used as clue. Doing so is shown in [Baumeister 00], for example.

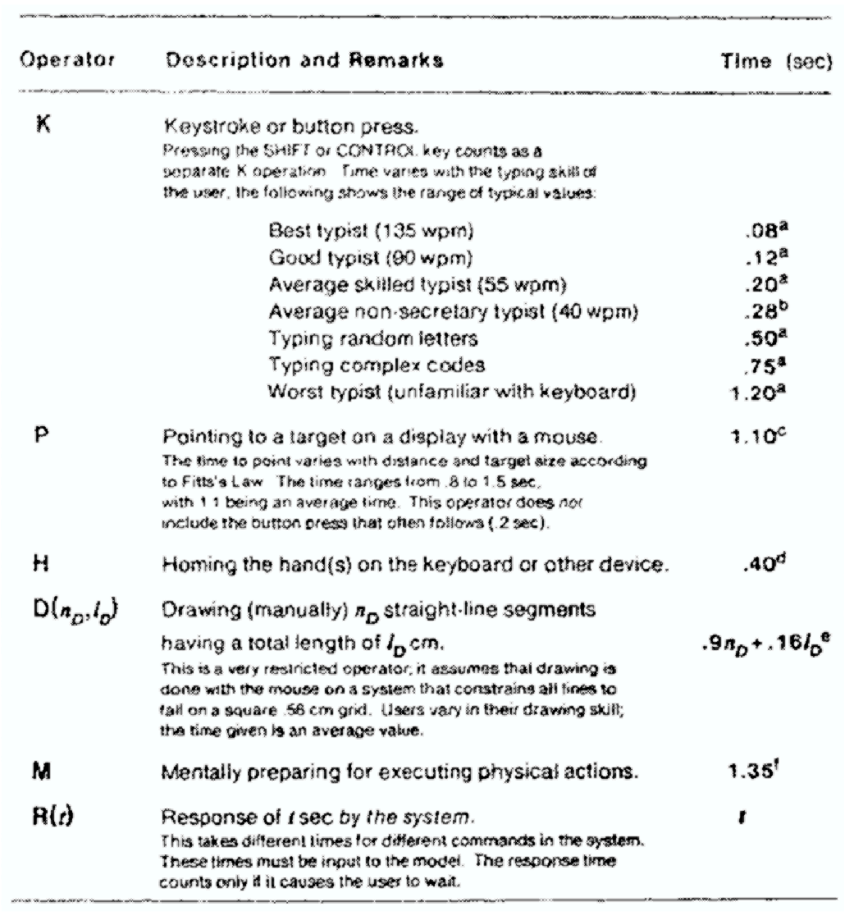
1.3.2. The Keystroke Level Model

The Keystroke Level Model (KLM) was introduced by Stuart K. Card, Thomas P. Moran, and Allen Newell in 1980 for comparison of the use of software with different interaction styles [Card 80]. Although it was developed earlier than GOMS, it now is regarded as the simplest GOMS modelling technique.

It is based on a simple serial stage model of human cognition and gives an estimate about the expert user's execution time of a task. The lack of a goal structure limits KLM to low level task representation.

1.3.2.1. KLM Operators

There are three types of operators in KLM. The physical motor operators represent acts like typing a key on the keyboard, or pointing to a target on the screen with a mouse. Cognitive operations are represented by the only mental operator M that stands for the preparing for one or more physical actions. The system response times are covered by the R operator. All KLM operators are listed in Figure 1.6.



Operator	Description and Remarks	Time (sec)
K	Keystroke or button press. Pressing the SHIFT or CONTROL key counts as a separate K operation. Time varies with the typing skill of the user, the following shows the range of typical values:	
	Best typist (135 wpm)	.08[a]
	Good typist (90 wpm)	.12[a]
	Average skilled typist (55 wpm)	.20[a]
	Average non-secretary typist (40 wpm)	.28[b]
	Typing random letters	.50[a]
	Typing complex codes	.75[a]
	Worst typist (unfamiliar with keyboard)	1.20[a]
P	Pointing to a target on a display with a mouse. The time to point varies with distance and target size according to Fitts's Law. The time ranges from .8 to 1.5 sec, with 1.1 being an average time. This operator does *not* include the button press that often follows (.2 sec).	1.10[c]
H	Homing the hand(s) on the keyboard or other device.	.40[d]
$D(n_D, l_D)$	Drawing (manually) n_D straight-line segments having a total length of l_D cm. This is a very restricted operator; it assumes that drawing is done with the mouse on a system that constrains all lines to fall on a square .56 cm grid. Users vary in their drawing skill; the time given is an average value.	$.9n_D + .16l_D$[e]
M	Mentally preparing for executing physical actions.	1.35[f]
$R(t)$	Response of t sec *by the system.* This takes different times for different commands in the system. These times must be input to the model. The response time counts only if it causes the user to wait.	t

Figure 1.6.: KLM Operators [Card 80].

1.3.2.2. Execution

A task is deconstructed into a list of single KLM operators. The required data can be obtained from sources such as specifications, handbooks, or through observation. The operators have particular execution times, which are added up to get the execution time of a task. For a detailed explanation of the method look in [Card 80].

1.3.2.3. Application

The missing ability to represent higher level goals limits KLM to use in elementary interface design and simple time predictions. At this low level of analysis its predictions are very accurate and therefore usable. KLM is easy to understand and models can be built in short time. In [Raskin 00] a detailed description of how KLM is used for simple input dialogue design of a temperature converter is given.

1.3.3. Natural GOMS Language

The Natural GOMS Language (NGOMSL) was introduced by David E. Kieras for user interface design and as a higher level notation for Cognitive Complexity Theory models in 1988 [Kieras 88].

NGOMSL is a structured natural-language notation for GOMS models with advanced support for cognition. Its serial stage architecture is suitable for hierarchical and sequential method modelling. As GOMS models, NGOMSL models enable the analyst to make quantitative and qualitative predictions of single expert user interaction with passive systems and active systems. In addition, it supports in checking for functional consistency and in making learning time predictions.

An NGOMSL model example of the Apple Macintosh Finder used for typical file manipulation tasks is given in Figure 1.7. A detailed description of NGOMSL and its application can be found in [Kieras 88, Kieras 96, Kieras 97a].

1.3.3.1. Cognitive Complexity Theory

The Cognitive Complexity Theory (CCT) was introduced by David E. Kieras and Peter G. Polson in 1985 for user complexity analysis by example of word processor use [Kieras 85].

The basic concept of CCT models is to incorporate knowledge of how a user accomplishes a goal, and how he does this with a specific device. Therefore CCT models consist of two parts, the description of the device, and the description of how to use it to accomplish a certain task.

```
Method for goal: delete a file.
Step 1. Accomplish goal: drag file to trash.
Step 2. Return with goal accomplished.

Method for goal: move a file.
Step 1. Accomplish goal: drag file to destination.
Step 2. Return with goal accomplished.

Method for goal: delete a directory.
Step 1. Accomplish goal: drag directory to trash.
Step 2. Return with goal accomplished.

Method for goal: move a directory.
Step 1. Accomplish goal: drag directory to destination.
Step 2. Return with goal accomplished.

Method for goal: delete an object.
Step 1. Accomplish goal: drag object to trash.
Step 2. Return with goal accomplished.

Method for goal: move an object.
Step 1. Accomplish goal: drag object to destination.
Step 2. Return with goal accomplished.

Method for goal: drag item to destination.
Step 1. Locate icon for item on screen.
Step 2. Move cursor to item icon location.
Step 3. Hold mouse button down.
Step 4. Locate destination icon on screen.
Step 5. Move cursor to destination icon.
Step 6. Verify that destination icon is reverse-video.
Step 7. Release mouse button.
Step 8. Return with goal accomplished.
```

Figure 1.7.: NGOMSL model example of the Apple Macintosh Finder used for file manipulation [Kieras 96].

The device description is done in Generalised Transition Networks (GTN), which are — as the name suggests — a more general form of transition networks. For information on transition networks in interface design see [Dix 98]. A production system is used to formalise the knowledge represented in a GOMS model.

CCT is described in detail in [Kieras 85], and in [Dix 98].

1.3.4. GOMS Language

Goals, Operators, Methods, and Selectors Language (GOMSL) was introduced by David E. Kieras for user interface design in 1995 [Kieras 95]. GOMSL is a formalized form of the Natural GOMS Language which can be executed with help of the GLEAN3 software tool.

A GOMSL model is made up of auditory and visual descriptions of the device, descriptions of the tasks to be accomplished, descriptions of the methods to achieve the tasks, and selection rules to choose the appropriate methods to accomplish those tasks.

A description of the auditory or visual characteristics of a device is done with objects with specific attributes one can hear when audible or see when looking at the object. Tasks are specified in so called task instances. These have attributes that specify the name of the task, its type, information required during execution of the task, and the next task to accomplish. In Figure 1.8 the description of the visual objects and tasks that are required by a GOMSL model version of the file manipulation example given in Figure 1.7 are shown.

A method to handle the task operation sequence must be included in the model. A selection rule is specified as if. . . then statements. The methods are specified by subgoals and operators. Figure 1.9 presents the GOMSL code required to define the selection rules and methods that are adequate to those in the NGOMSL model example in Figure 1.7.

1.3.4.1. The GLEAN3 Simulation Tool

The GOMS Language Evaluation and Analysis 3 (GLEAN3) tool is a GOMS model simulation tool that can process GOMSL. It is based on GLEAN, developed by Scott Wood in 1993, and was reimplemented by Anthony Hornof and David E. Kieras and Kasem Abotel and Scott D. Wood in 1995 [Kieras 95].

The main difference between GLEAN and GLEAN3 is that the latter is based on a simplified version of the EPIC architecture (Figure 1.10) and hence is more powerful and better defined than its predecessor. EPIC is an architecture for simulating human cognition and performance. See [Kieras 97b] for more information. GLEAN3 executables can be obtained from [Kieras 05] and run on the Macintosh and the Windows platform.

```
Define_model: "MacOS File Maintenance Example"
Starting_goal is Perform Disk_Maintenance.

Visual_object: Trash_can_icon
Label is Trash.

Visual_object: file1_icon
Label is "file1.txt".

Visual_object: file2_icon
Label is "file2.txt".

Visual_object: Documents_icon
Label is "Documents".

Visual_object: Work_icon
Label is "Work".

Visual_object: Temp_icon
Label is "Temp".

Task_item: T1
Name is First.
Type is delete_file.
Filename is "file1.txt".
Enclosing_directory is "Work".
Next is T2.

Task_item: T2
Name is T2.
Type is move_file.
Filename is "file2.txt".
Enclosing_directory is "Work".
Destination_directory is "Documents".
Next is T3.

Task_item: T3
Name is T3.
Type is delete_directory.
Directory_name is "Temp".
Next is T4.

Task_item: T4
Name is T4.
Type is move_directory.
Directory_name is "Work".
Destination_directory is "Documents".
Next is None.
```

Figure 1.8.: Visual object and task specification in GOMSL [Kieras 99].

```
Method_for_goal: Perform Disk_Maintenance
Step 1. Store First under <current_task_name>.
Step 2. Decide: If <current_task_name> is None,
        Then Delete <current_task>; Delete <current_task_name>;
        Return_with_goal_accomplished.
Step 3. Get_task_item_whose Name is <current_task_name>
        and_store_under <current_task>.
Step 3. Accomplish_goal: Perform Disk_Task.
Step 4. Store Next of <current_task> under <current_task_name>.
Step 5. Goto 2.

Selection_rules_for_goal: Perform Disk_Task
If Type of <current_task> is delete_file,
  Then Accomplish_goal: delete object using
        Filename of <current_task>.
If Type of <current_task> is move_file,
  Then Accomplish_goal: move object using
        Filename of <current_task>, and
        Destination_directory of <current_task>.
If Type of <current_task> is delete_directory,
  Then Accomplish_goal: delete object using
        Directory_name of <current_task>.
If Type of <current_task> is move_directory,
  Then Accomplish_goal: move object using
        Directory_name of <current_task>, and
        Destination_directory of <current_task>.
Return_with_goal_accomplished.

Method_for_goal: delete object using <object_name>
Step 1. Accomplish_goal: drag object using <object_name>, and Trash.
Step 2. Return_with_goal_accomplished.

Method_for_goal: move object using <object_name>, and
<destination_directory>
Step 1. Accomplish_goal: drag object using
        <object_name>, and <destination_directory>.
Step 2. Return_with_goal_accomplished.

Method_for_goal: drag object using <object>, <destination>
Step 1. Look_for_object_whose Label is <object>
        and_store_under <target>.
Step 2. Point_to <target>.
Step 3. Hold_down Mouse_Button.
Step 4. Look_for_object_whose Label is <destination>
        and_store_under <target>.
Step 5. Point_to <target>.
Step 6. Verify "icon is at destination".
Step 7. Release Mouse_Button.
Step 8. Delete <target>; Return_with_goal_accomplished.
```

Figure 1.9.: Selection rules and methods in GOMSL [Kieras 99].

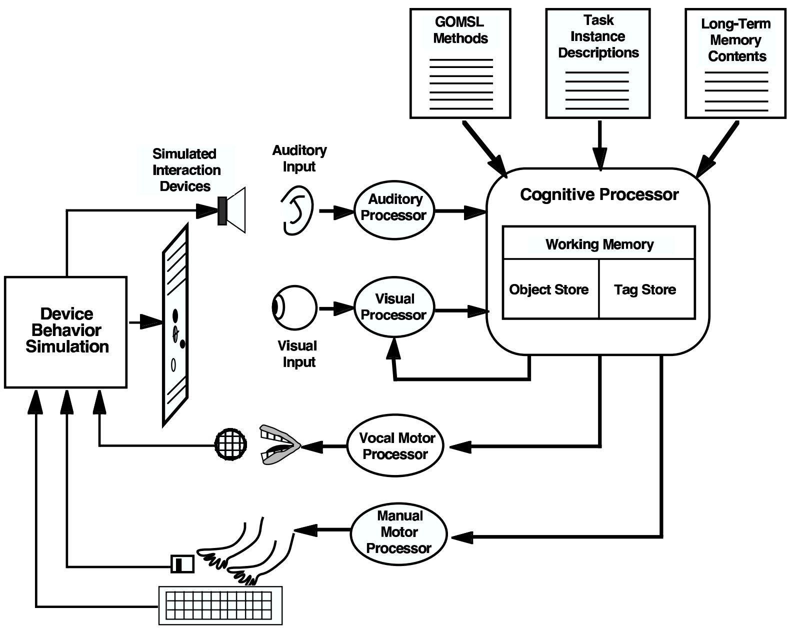


Figure 1.10.: GLEAN3 architecture [Kieras 99].

The GLEAN3 tool interprets GOMSL models and carries out those calculations done by hand when using other GOMS methods, and furthermore is able to make statements about working memory requirements and the learnability of a model. The output of GLEAN3 is a leaning analysis, the execution times, and an execution log. The learning analysis describes the learnability of the methods defined in the model in giving the number of operators necessary to accomplish each method, and the number of operators which can be learned from akin methods. The execution times given are the average and the total execution times of the specific methods, the proportion of total execution time of the specific methods and the total estimated execution time for the whole model.

Furthermore GLEAN3 produces a complete detailed protocol of the execution of the tasks specified in the model. This is useful for debugging the GOMSL model and makes it possible to verify the consistency of the model.

A detailed description of the use of GOMSL and GLEAN3 is given in [Kieras 99]. The use of GLEAN3 and GOMSL for user interface evaluation of military shipboard workstations is described in [Kieras 04].

1.3.5. Cognitive Perceptual Motor GOMS

Cognitive Perceptual Motor GOMS (CPM-GOMS) is the only GOMS method that allows for analysis of tasks running in parallel and was introduced by Bonnie E. John in 1988 [John 88]. It is based on the Critical Path Method, which is a network model for project management that can deal with sequential and parallel tasks. Therefore CPM-GOMS is also referred to as Critical Path Method GOMS.

Originally CPM-GOMS models were build with project management tools and presented using PERT (Program Evaluation and Review Technique) charts. The procedure to do so was nearly unspecified. In Project Ernestine (see [Gray 92]), it was successfully used to evaluate a new toll and assistance operator's workstation design at the NYNEX telephone company and helped the company to save almost $2 million dollars a year. Figure 1.11 shows a CPM-GOMS model of the old workstation use.

CPM-GOMS is the only GOMS method that is incorporates all aspects of the MHP.

1.3.5.1. Automation with Apex

Apex is a cognitive modelling tool developed by Michael Freed at the NASA Ames Research Center [Freed 04]. It has a complex human operator model that has the ability to cope with limited cognitive, perceptual, and motor resources. Templates of human skills are integrated.

GOMSL models can be written alike GOMSL descriptions in PDL (Procedure Description Language), which is the programming language of Apex. With the help of Sherpa — a front end for Apex — it is then possible to create PERT charts of the models.

For more information on this topic have a look at [John 02, Freed 03].

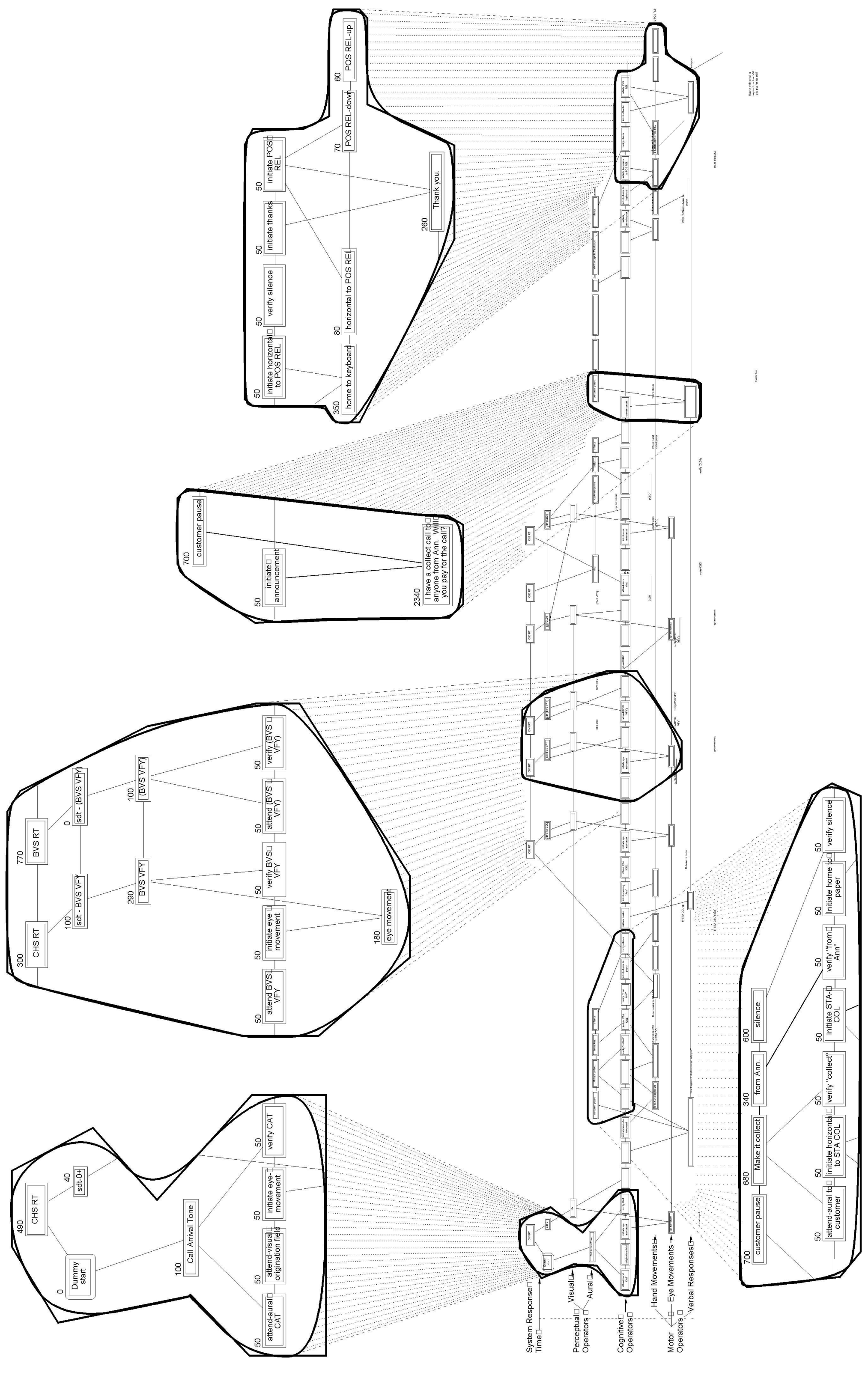


Figure 1.11.: A CPM-GOMS model of a toll and assistance operator's workstation use [John 95b].

2. Hierarchical Task Analysis of Performing a WBA

To learn what people are actually doing and what they might want to do during a WBA, and to make sure that all relevant aspects are incorporated in the analysis of YB-Edit use, I analysed the whole process of doing a WBA with the HTA method at first. The tasks that incorporate the use of YB-Edit are used later as high level goals for a GOMSL analysis to break the users interaction with YB-Edit down to basic operations. This approach avoids falling into the trap of defining the goal hierarchy of the use of YB-Edit by the interface of YB-Edit and provides a thorough overview of doing a WBA for further integration of those parts of WBA, which are not yet covered by YB-Edit. Differences in the outperforming of the WBA method that might influence the use of YB-Edit are included this way, too.

The level of analysis is that of higher level goals, since low level goals are to be analysed later with the GOMSL analysis method.

2.1. Data Acquisition

To obtain the required information I made interviews with all people who are experienced in WBA that I could reach. The interviewees are all members of the RVS Group (German: RVS - Rechnernetze und Verteilte Systeme [Ladkin 05]), have done several WBAs and are in part involved in teaching the method in seminars. The basic question I asked was how they perform a WBA.

To record the information, I designed record sheets in the style of planning sheets (see Appendix A), with a field for the task name and number and a table with two columns, one for the subtasks and one for comments. Every Task that was decomposed was noted on a new record sheet.

2.1.1. Data Preparation for Presentation

I used an outlining tool to collect the information acquired in the interviews. Because they are easier to understand than the tables, I chose a hierarchical diagram form for representation of the results of the analysis.

To draw HTA diagrams, I wrote a tool, which included the layout information to gain a HTA diagram layout close to the standard. I used the tool to convert the textual representation of the data to a format understood by a graph drawing tool, which then processed the graphical representations.

Since the resulting HTA diagram measures 2.8 meters in width when printed with a readable font size, I decided to split the diagram up in sub diagrams.

2.2. Performing a WBA

The performance of a WBA (Figure 2.1) always starts with the gathering of information. The information is then arranged either in a Why-Because List, or a in List of Facts, and optionally an auxiliary List of Facts (e.g. lists in chronological order) are made. When this is done, the construction of the WB Graph begins with the determination of the top node — the accident of the analysed case. Then, the necessary causal factors are determined and the WB Graph is corrected. This is represented as two sequent tasks even though there is not necessarily a chronology, as stated in the corresponding plan (plan 0 in Figure 2.1). Finally the analyst writes the report.

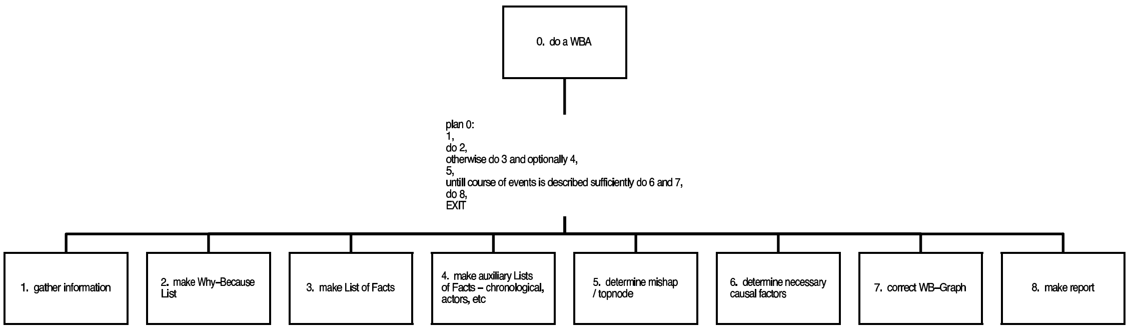

0. do a WBA
plan 0:
1,
do 2,
otherwise do 3 and optionally 4,
5,
untill course of events is described sufficiently do 6 and 7,
do 8,
EXIT
1. gather information
2. make Why–Because List
3. make List of Facts
4. make auxiliary Lists of Facts – chronological, actors, etc
5. determine mishap / topnode
6. determine necessary causal factors
7. correct WB–Graph
8. make report

Figure 2.1.: Performing a WBA.

2.2.1. Gather Information

The gathering of information is done in a quite common way. First the sources of information have to be identified and then the information material has to be obtained. The information is then sighted and selected. See Figure 2.2 for details.

2.2.2. Presorting of Factors

All interviewees extract the facts and arrange them. I observed two basic ways to do so.

2.2.2.1. Make a List of Facts

A List of Facts is a collection of all facts found in the information material, that are relevant to the accident. To obtain these facts, the information material is again read in-depth and the detected facts are collected with reference number with help of a text editor (see Figure 2.3).

2.2.2.2. Make a Why-Because List

A Why-Because List incorporates information about the facts and their relations to each other. Therefore in difference to making a List of Facts, where the facts are only collected, the relations between the facts are expressed as Why-Because pairs of facts, that are recorded in a text editor. When this variant is used, the counterfactual test is applied and a consistency check done at this early stage of the WBA. See Figure 2.4.

2.2.3. Make an Auxiliary List of Facts

Auxiliary Lists of Facts are Lists of Facts where the facts are arranged in a specific manner. They are optional, but often help to gather a better understanding of the incident. First of all the classification system is laid down, then all facts are sighted again and written down according to the classification using a text editor. It happens, that the classification system chosen did not cover all facts, so that the whole process has to be redone. The process is described in Figure 2.5.

2.2.4. Determine the Mishap

To spot the fact that makes the accident is the first step in developing a WB Graph. Therefore the facts are reviewed and assessed. Doing so in a team is of advantage. When the mishap is spotted, it is added as top node of the new WB Graph using YB-Edit. See Figure 2.6.

2.2.5. Determine the Necessary Causal Factors

To determine the necessary causal factors, the facts are reviewed first. What is done next is determined by the way the facts were arranged. If the facts are available as Why-Because List, the necessary information can be directly read, since the facts are arranged in Why-Because pairs. After viewing the facts the causal factors can be added to the graph.

If the List of Facts is utilised, the facts must be assessed and the necessary causal factors must be spotted, before they can be added to the graph with a reference to the List of Facts entry. See Figure 2.7.

2.2.6. Correct the WB Graph

The graph is corrected while the necessary causal factors are determined. Sometimes the addition of a necessary causal factor causes inconsistencies in the graph, that must be corrected and thus necessary causal factors are added while the graph is in correction. Since the correction of the graph is not part of determining a causal factor, nor the graph is just corrected — what may incorporate a determination of a necessary causal factor on occasion — after determining the top node, the determining of necessary causal factors and the correction are represented as separate high level tasks.

To correct the WB Graph (see Figure 2.8), it is printed and viewed at, then the causalities in the graph are assessed. To check the correctness of the causal relations, the counterfactual test is applied. If considered necessary, the graph is changed. These changes can be the addition of a necessary causal factor or node (Figure 2.9), the removal of a node (Figure 2.10), merging of nodes (Figure 2.11), splitting of nodes (Figure 2.12), changing of edges (Figure 2.13), or node descriptions (Figure 2.14). Note that many of these changes require the adaption of the List of Facts.

2.2.7. Make the Report

The last step of a WB-Analysis is the making of the report. The report comprises an overview, an abstract, the operations and results of the analysis, a comparison of the conclusions of an external report and the conclusions of the WBA, and a discussion of any differences between them. See Figure 2.15.

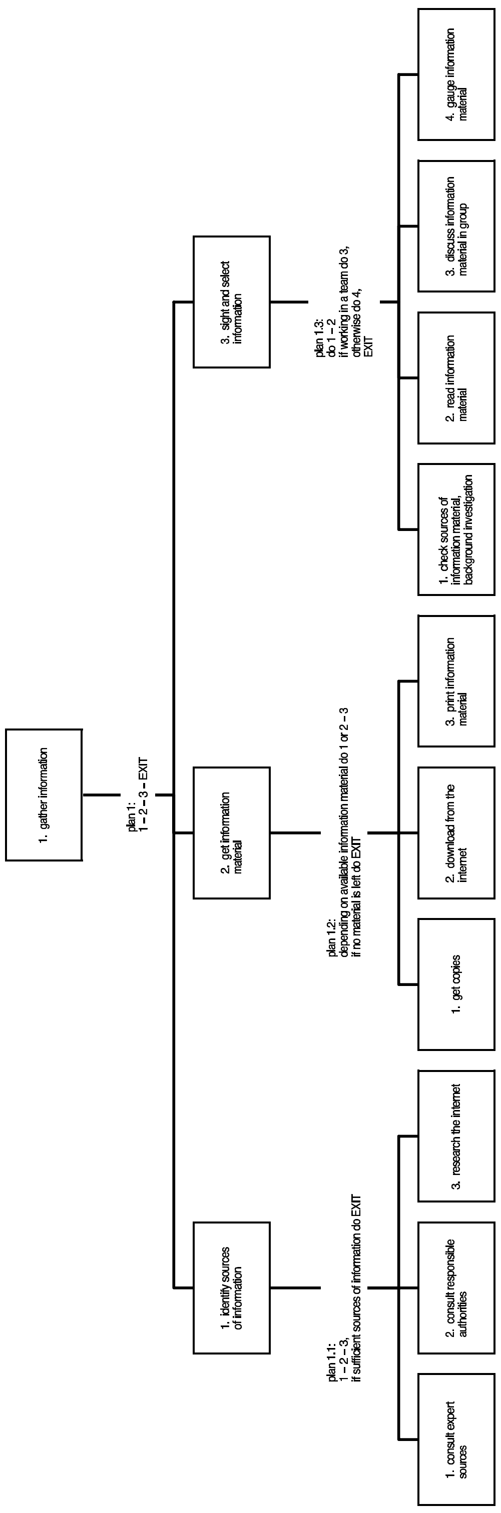

1. gather information
plan 1:
1 – 2 – 3 – EXIT
1. identify sources of information
2. get information material
3. sight and select information
plan 1.1:
1 – 2 – 3,
if sufficient sources of information do EXIT
1. consult expert sources
2. consult responsible authorities
3. research the internet
plan 1.2:
depending on available information material do 1 or 2 – 3
if no material is left do EXIT
1. get copies
2. download from the internet
3. print information material
plan 1.3:
do 1 – 2
if working in a team do 3,
otherwise do 4,
EXIT
1. check sources of information material, background investigation
2. read information material
3. discuss information material in group
4. gauge information material

Figure 2.2.: Gathering the information.

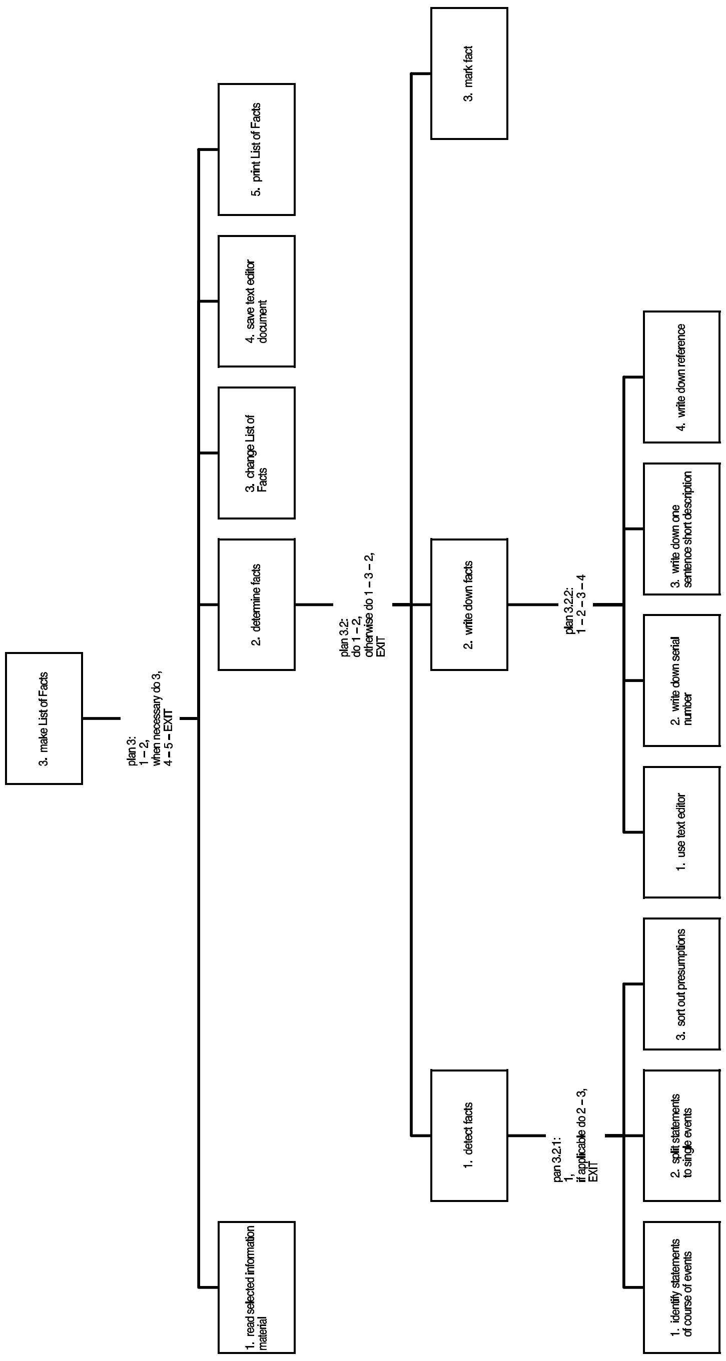


Figure 2.3.: Making a List of Facts.

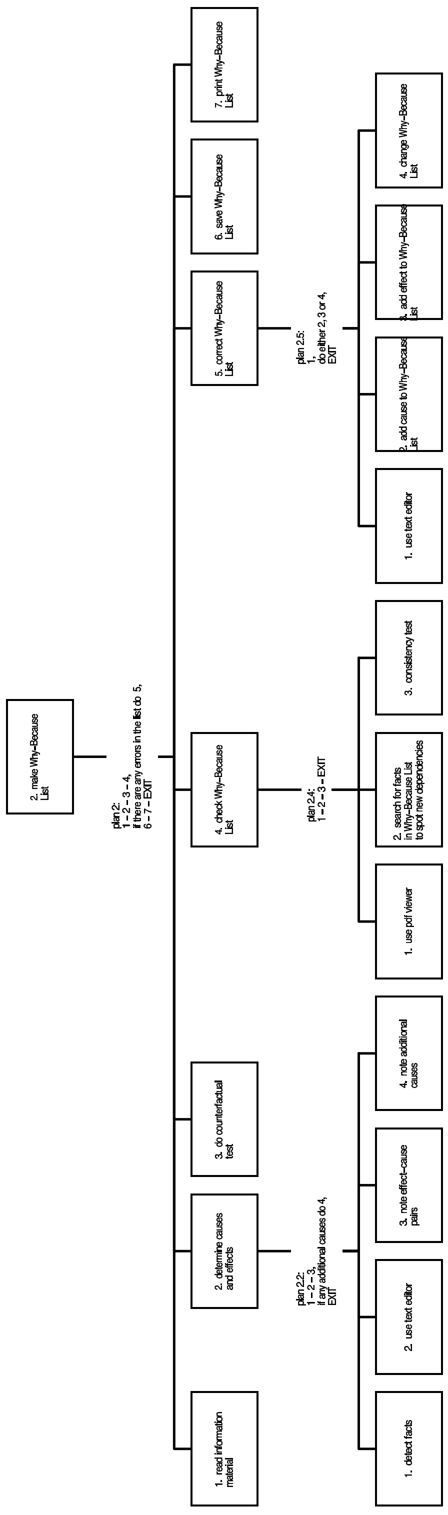

2. make Why-Because List
plan 2: 1 – 2 – 3 – 4, if there are any errors in the list do 5, 6 – 7 – EXIT
1. read information material
2. determine causes and effects
3. do counterfactual test
4. check Why-Because List
5. correct Why-Because List
6. save Why-Because List
7. print Why-Because List
plan 2.2: 1 – 2 – 3, if any additional causes do 4, EXIT
1. detect facts
2. use text editor
3. note effect–cause pairs
4. note additional causes
plan 2.4: 1 – 2 – 3 – EXIT
1. use pdf viewer
2. search for facts in Why-Because List to spot new dependencies
3. consistency test
plan 2.5: 1, do either 2, 3 or 4, EXIT
1. use text editor
2. add cause to Why-Because List
3. add effect to Why-Because List
4. change Why-Because List

Figure 2.4.: Making a Why-Because List.

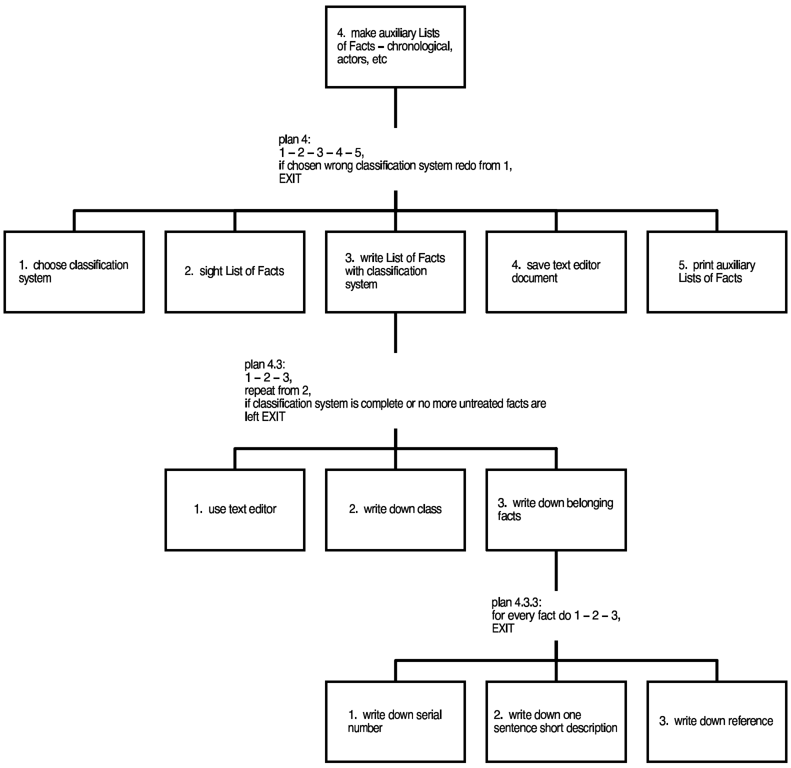

4. make auxiliary Lists of Facts – chronological, actors, etc
plan 4:
1 – 2 – 3 – 4 – 5,
if chosen wrong classification system redo from 1,
EXIT
1. choose classification system
2. sight List of Facts
3. write List of Facts with classification system
4. save text editor document
5. print auxiliary Lists of Facts
plan 4.3:
1 – 2 – 3,
repeat from 2,
if classification system is complete or no more untreated facts are left EXIT
1. use text editor
2. write down class
3. write down belonging facts
plan 4.3.3:
for every fact do 1 – 2 – 3,
EXIT
1. write down serial number
2. write down one sentence short description
3. write down reference

Figure 2.5.: Making an auxiliary List of Facts.

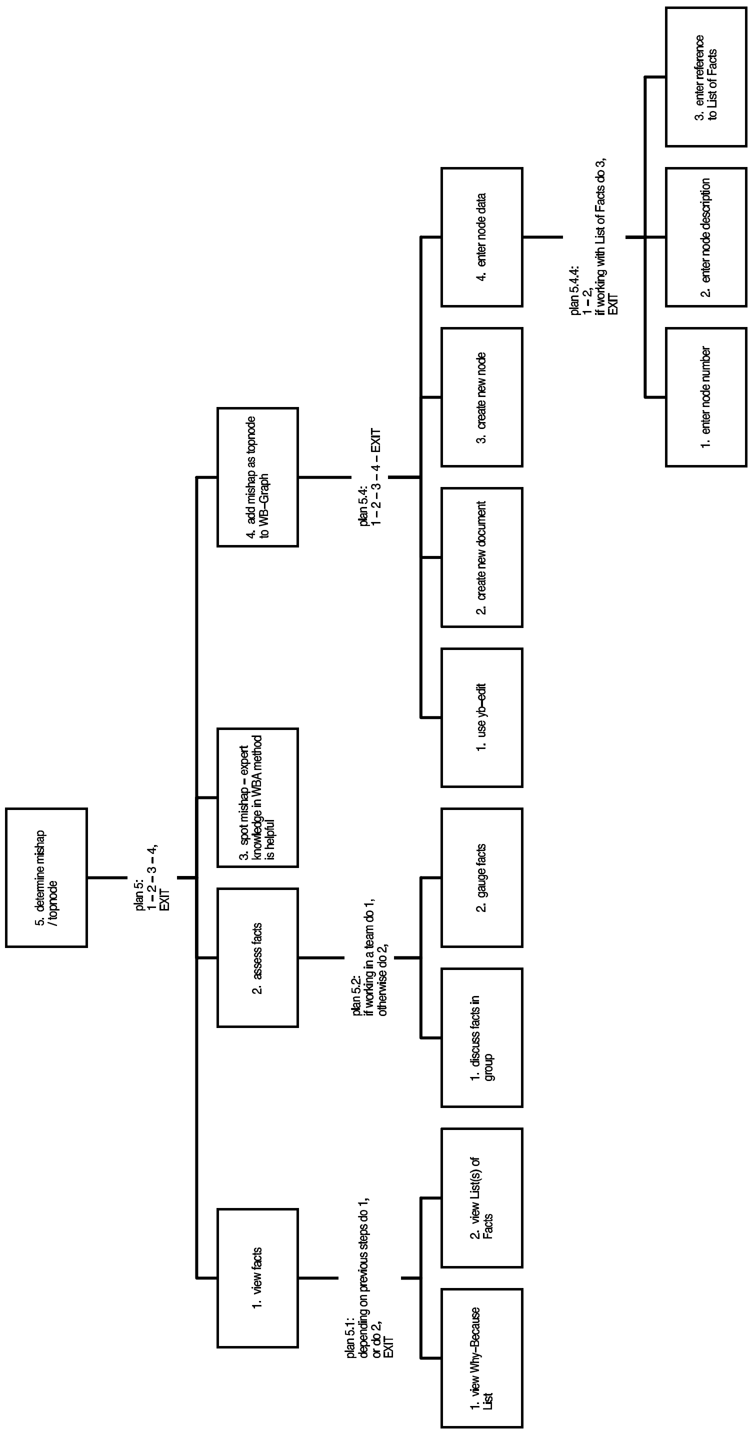

5. determine mishap / topnode
plan 5: 1 – 2 – 3 – 4, EXIT
1. view facts
2. assess facts
3. spot mishap – expert knowledge in WBA method is helpful
4. add mishap as topnode to WB–Graph
plan 5.1: depending on previous steps do 1, or do 2, EXIT
1. view Why–Because List
2. view List(s) of Facts
plan 5.2: if working in a team do 1, otherwise do 2,
1. discuss facts in group
2. gauge facts
plan 5.4: 1 – 2 – 3 – 4 – EXIT
1. use yb–edit
2. create new document
3. create new node
4. enter node data
plan 5.4.4: 1 – 2, if working with List of Facts do 3, EXIT
1. enter node number
2. enter node description
3. enter reference to List of Facts

Figure 2.6.: Determining the mishap.

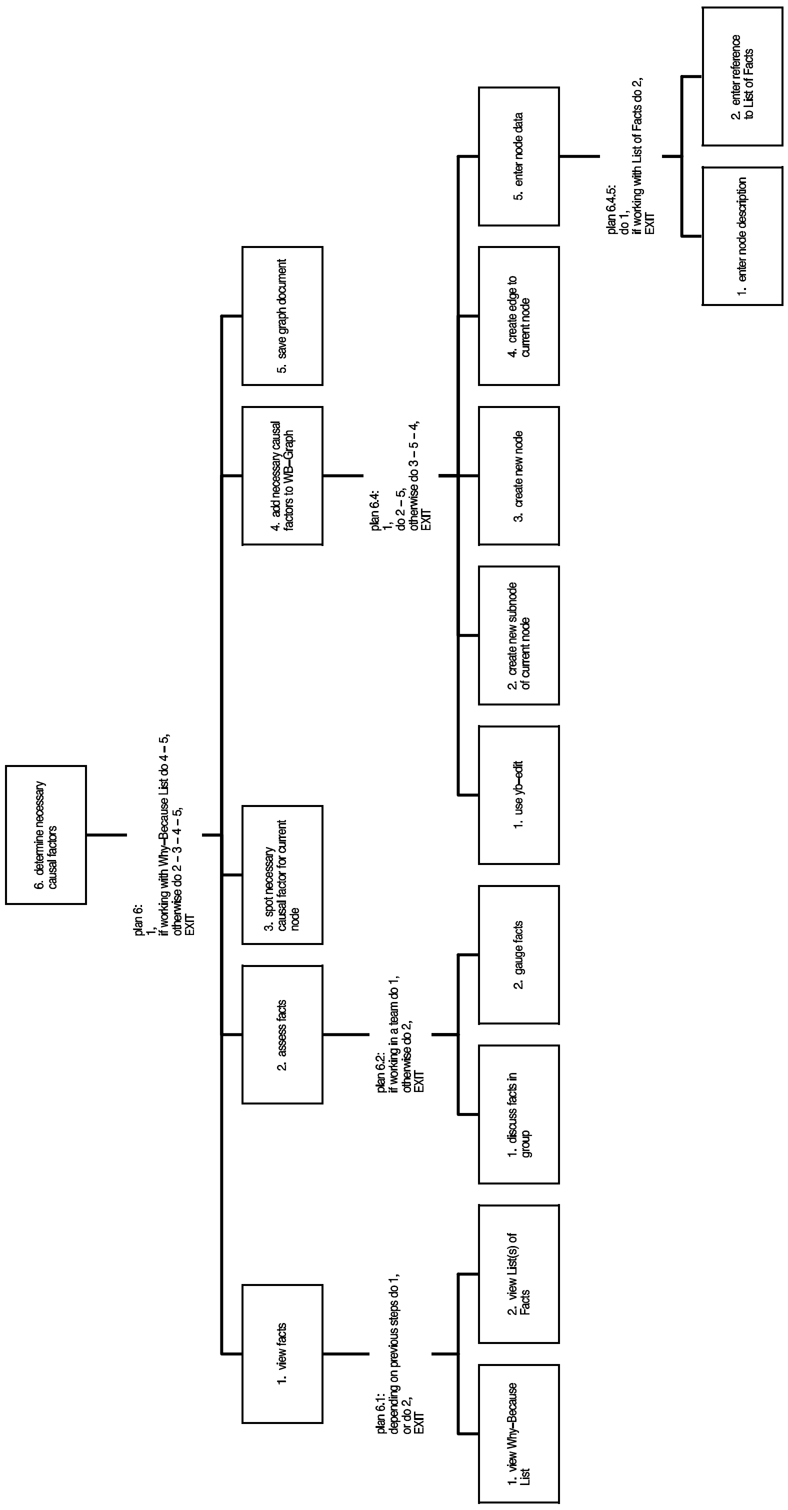

6. determine necessary causal factors
plan 6:
1,
if working with Why–Because List do 4 – 5,
otherwise do 2 – 3 – 4 – 5,
EXIT
1. view facts
2. assess facts
3. spot necessary causal factor for current node
4. add necessary causal factors to WB–Graph
5. save graph document
plan 6.1:
depending on previous steps do 1,
or do 2,
EXIT
1. view Why–Because List
2. view List(s) of Facts
plan 6.2:
if working in a team do 1,
otherwise do 2,
EXIT
1. discuss facts in group
2. gauge facts
plan 6.4:
1,
do 2 – 5,
otherwise do 3 – 5 – 4,
EXIT
1. use yb–edit
2. create new subnode of current node
3. create new node
4. create edge to current node
5. enter node data
plan 6.4.5:
do 1,
if working with List of Facts do 2,
EXIT
1. enter node description
2. enter reference to List of Facts

Figure 2.7.: Determining the necessary causal factors.

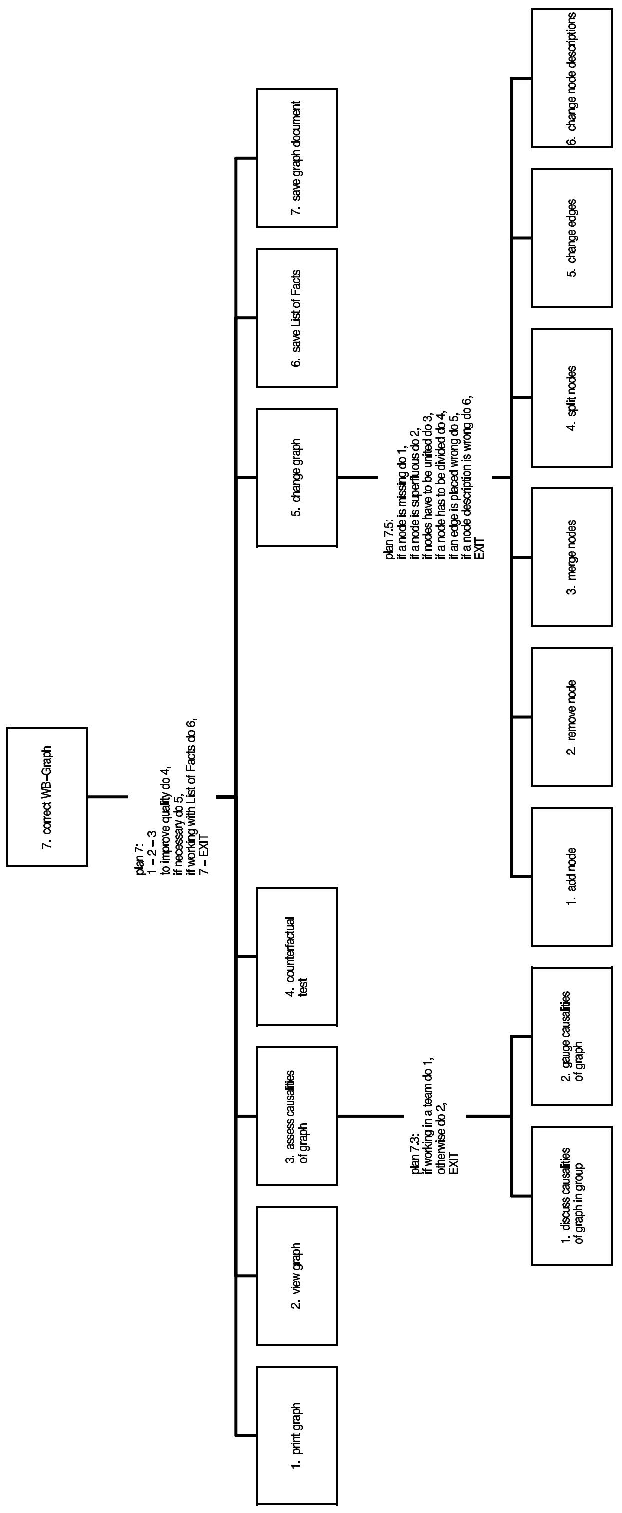

7. correct WB-Graph
plan 7:
1 – 2 – 3
to improve quality do 4,
if necessary do 5,
if working with List of Facts do 6,
7 – EXIT
1. print graph
2. view graph
3. assess causalities of graph
4. counterfactual test
5. change graph
6. save List of Facts
7. save graph document
plan 7.3:
if working in a team do 1,
otherwise do 2,
EXIT
1. discuss causalities of graph in group
2. gauge causalities of graph
plan 7.5:
if a node is missing do 1,
if a node is superfluous do 2,
if nodes have to be united do 3,
if a node has to be divided do 4,
if an edge is placed wrong do 5,
if a node description is wrong do 6,
EXIT
1. add node
2. remove node
3. merge nodes
4. split nodes
5. change edges
6. change node descriptions

Figure 2.8.: Correction of the WB Graph

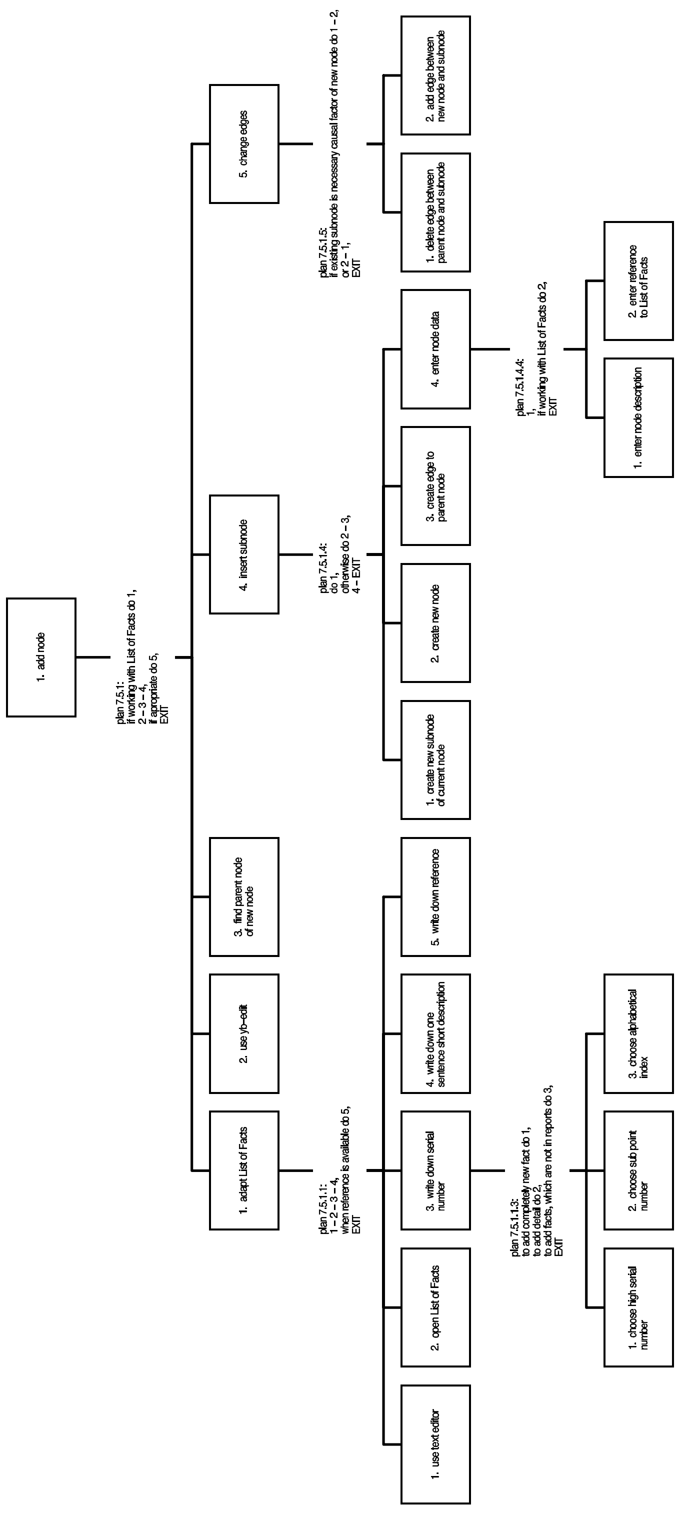

1. add node
plan 7.5.1: if working with List of Facts do 1, 2 – 3 – 4, if appropriate do 5, EXIT
1. adapt List of Facts
2. use yb–edit
3. find parent node of new node
4. insert subnode
5. change edges
plan 7.5.1.1: 1 – 2 – 3 – 4, when reference is available do 5, EXIT
1. use text editor
2. open List of Facts
3. write down serial number
4. write down one sentence short description
5. write down reference
plan 7.5.1.1.3: to add completely new fact do 1, to add detail do 2, to add facts, which are not in reports do 3, EXIT
1. choose high serial number
2. choose sub point number
3. choose alphabetical index
plan 7.5.1.4: do 1, otherwise do 2 – 3, 4 – EXIT
1. create new subnode of current node
2. create new node
3. create edge to parent node
4. enter node data
plan 7.5.1.4.4: 1, if working with List of Facts do 2, EXIT
1. enter node description
2. enter reference to List of Facts
plan 7.5.1.5: if existing subnode is necessary causal factor of new node do 1 – 2, or 2 – 1, EXIT
1. delete edge between parent node and subnode
2. add edge between new node and subnode

Figure 2.9.: Adding a node to the graph.

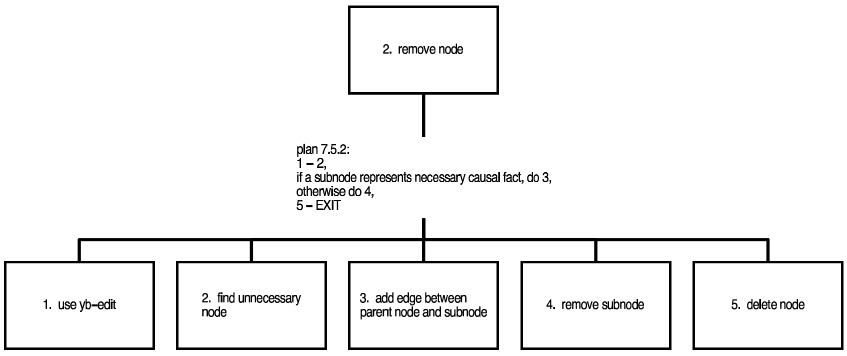


Figure 2.10.: Removing a node from the graph.

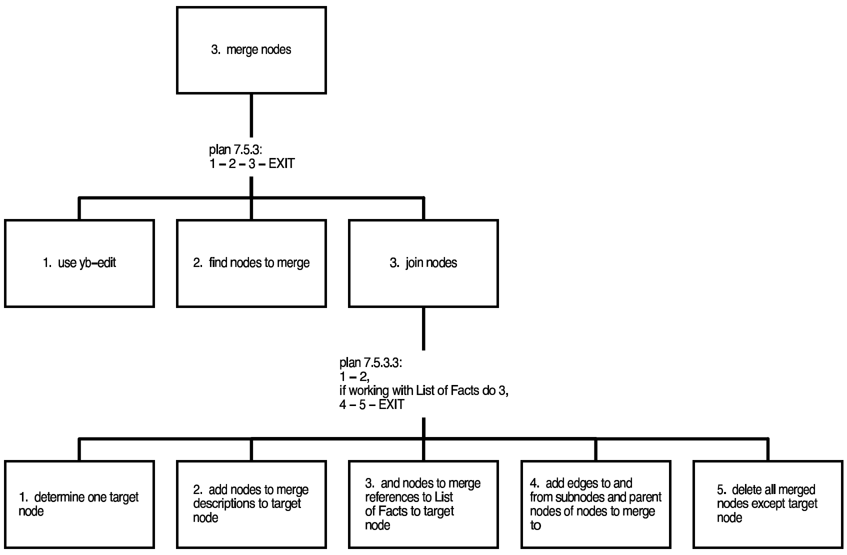

3. merge nodes
plan 7.5.3:
1 – 2 – 3 – EXIT
1. use yb–edit
2. find nodes to merge
3. join nodes
plan 7.5.3.3:
1 – 2,
if working with List of Facts do 3,
4 – 5 – EXIT
1. determine one target node
2. add nodes to merge descriptions to target node
3. and nodes to merge references to List of Facts to target node
4. add edges to and from subnodes and parent nodes of nodes to merge to
5. delete all merged nodes except target node

Figure 2.11.: Merging nodes.

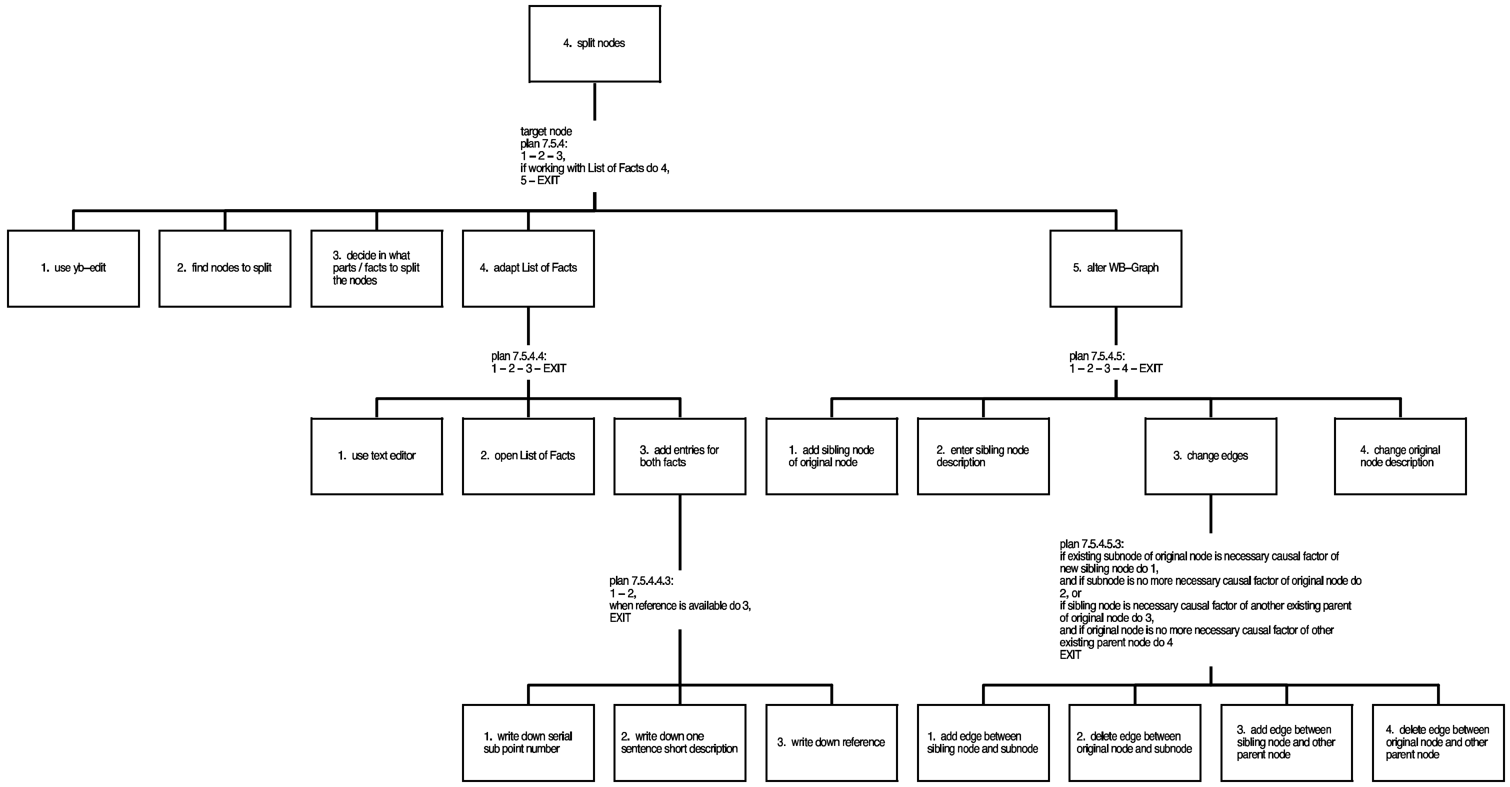

4. split nodes
target node
plan 7.5.4:
1 – 2 – 3,
if working with List of Facts do 4,
5 – EXIT
1. use yb–edit
2. find nodes to split
3. decide in what parts / facts to split the nodes
4. adapt List of Facts
5. alter WB–Graph
plan 7.5.4.4:
1 – 2 – 3 – EXIT
1. use text editor
2. open List of Facts
3. add entries for both facts
plan 7.5.4.4.3:
1 – 2,
when reference is available do 3,
EXIT
1. write down serial sub point number
2. write down one sentence short description
3. write down reference
plan 7.5.4.5:
1 – 2 – 3 – 4 – EXIT
1. add sibling node of original node
2. enter sibling node description
3. change edges
4. change original node description
plan 7.5.4.5.3:
if existing subnode of original node is necessary causal factor of new sibling node do 1,
and if subnode is no more necessary causal factor of original node do 2, or
if sibling node is necessary causal factor of another existing parent of original node do 3,
and if original node is no more necessary causal factor of other existing parent node do 4
EXIT
1. add edge between sibling node and subnode
2. delete edge between original node and subnode
3. add edge between sibling node and other parent node
4. delete edge between original node and other parent node

Figure 2.12.: Splitting a node.

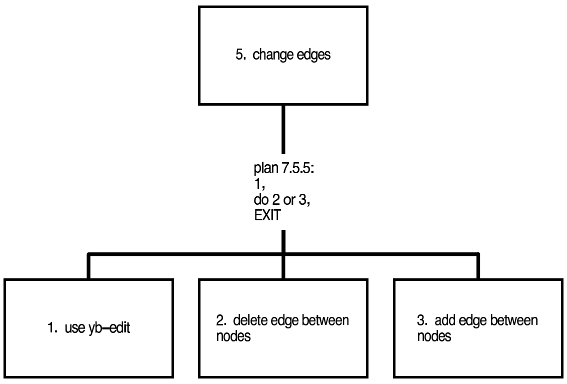


Figure 2.13.: Changing edges between nodes.

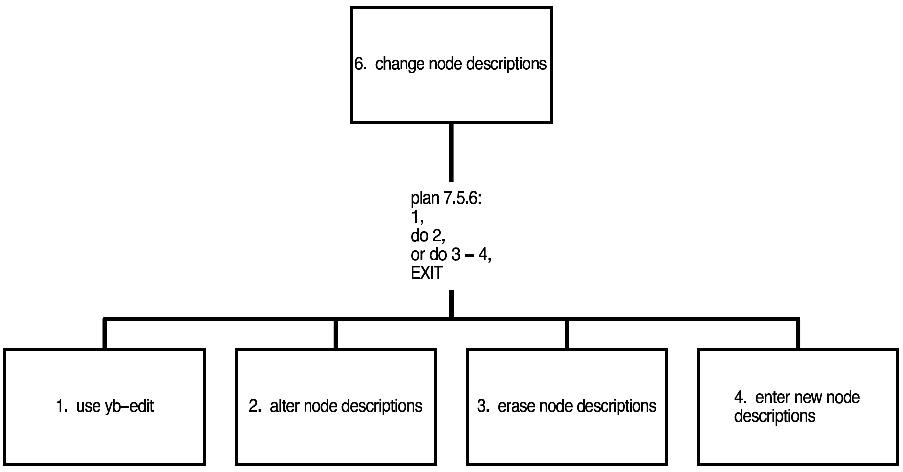

6. change node descriptions
plan 7.5.6:
1,
do 2,
or do 3 – 4,
EXIT
1. use yb–edit
2. alter node descriptions
3. erase node descriptions
4. enter new node descriptions

Figure 2.14.: Changing node descriptions

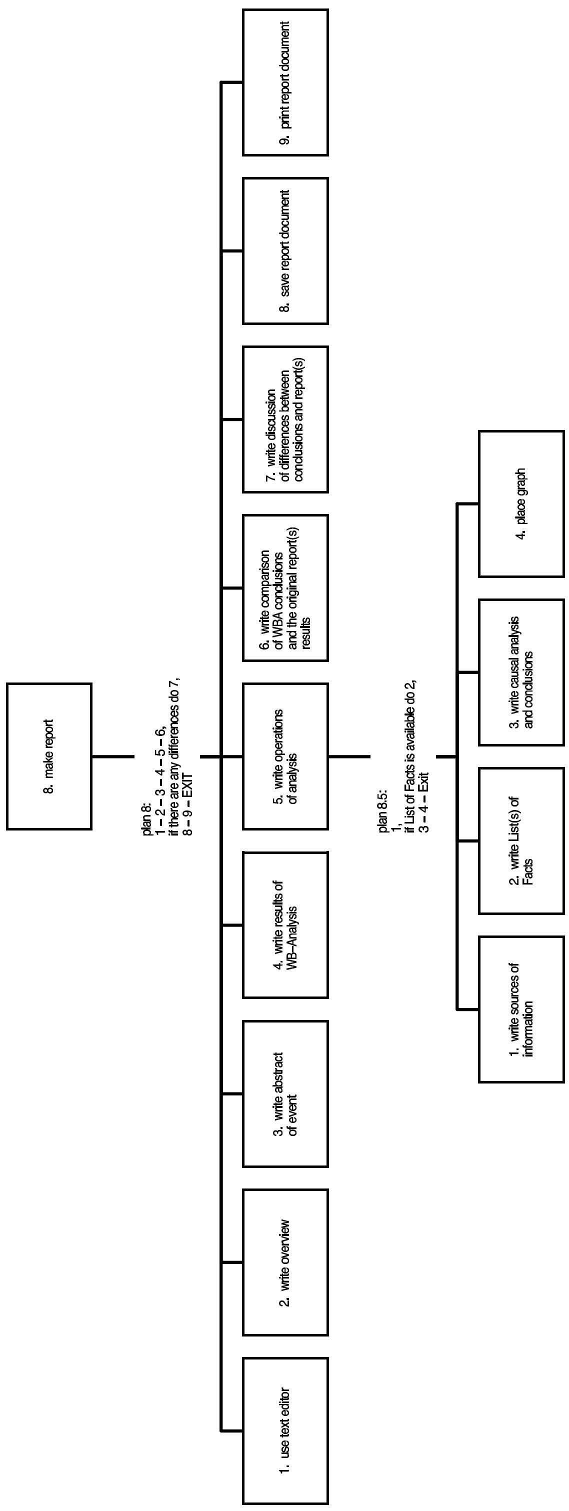

8. make report
plan 8:
1 – 2 – 3 – 4 – 5 – 6,
if there are any differences do 7,
8 – 9 – EXIT
1. use text editor
2. write overview
3. write abstract of event
4. write results of WB–Analysis
5. write operations of analysis
6. write comparison of WBA conclusions and the original report(s) results
7. write discussion of differences between conclusions and report(s)
8. save report document
9. print report document
plan 8.5:
1,
if List of Facts is available do 2,
3 – 4 – Exit
1. write sources of information
2. write List(s) of Facts
3. write causal analysis and conclusions
4. place graph

Figure 2.15.: Making the report.

2.3. Results

With the help of the HTA the high level goals of YB-Edit use for the creation and manipulation of a WB Graph are determined. These goals are easy to identify by looking for tasks that affect the manipulation of the WB Graph in direct and therefore incorporate "use yb-edit" on their first lower hierarchy level.

High Level Goals of YB-Edit Use

- add a (top-)node to the graph (Figure 2.6)
- add a necessary causal factor to the graph (Figure 2.7)
- add a node in the graph (insert) (Figure 2.9)
- remove a node (Figure 2.10)
- change an edge between nodes (Figure 2.13)
- change the node description (Figure 2.14)
- merge nodes (Figure 2.11)
- split nodes (Figure 2.12)

The analysis went deeper than originally planed. This happened because what a top level goal of YB-Edit use is was not transparent during the interviews. The information gained illuminate the basic structure of how the tasks are accomplished and give a good idea of how the users of YB-Edit perceive the program.

Furthermore the analysis displays two different strategies in doing a WBA. One strategy is to arrange the facts in a List of Facts, an approach established since the analysis of the "Royal Majesty" incident [Ladkin 03], in which the facts are collected in a list that can be referenced later. In this approach, the causal relations are established while building the WB Graph (see Figure 2.3 and 2.8). The other strategy is to arrange the facts in a Why-Because List, that require the analyst to establish the causal relations earlier in the process of a WBA (see Figure 2.4). See [Sieker 04] for further details.

Both strategies should be supported by a new design for the interface of YB-Edit.

3. The Analysis of YB-Edit

3.1. The Topics Analysed

There is a wide range of topics, which are either too obvious, very well described elsewhere, or just not ascertainable with task analysis methods, that I did not make them topic of my analysis. These are topics like how to arrange the program menu, the enhancements of providing an undo function, or the benefits of seeing the whole graph at once or having a map of the graph for better orientation.

Useful information on GUI design regarding the use of standard components like widgets and menus can be found e.g. in [Cooper 95] or [Raskin 00]. And of course it would be good in any way to have an undo function, a really big screen, or at least an orientation guide to navigate the WB Graph.

What is analysed is the direct manipulation of the representation of the graph by the user. The way the user is able to create and alter a WB Graph using the mouse as input device is what makes YB-Edit special. The Goal of this analysis is to detect possible improvement opportunities or sources of error, that may have found their way into YB-Edit during its evolution.

3.2. Why GOMS Language?

GOMSL has the highest formality and precision of all GOMS variants and the required calculations can be automated with help of GLEAN3. It supports in determination of the functional consistency of a model, which is of advantage when optimising individual methods.

GLEAN3 provides the analyst with detailed time predictions of single method execution, overall single method execution, and the whole process execution of an expert user.

The ability to predict learning time is a GOMSL exclusive feature, even though only relative comparisons of competing design models can be made.

3.3. The User Interface of YB-Edit

YB-Edit's GUI is shown in Figure 3.1. It consists of a menu bar at the top of the window, a canvas where the WB Graph is drawn at the left, and a display list at the right. The graph-related commands are executed through pop-up menus. The elements of the graph — the nodes and the edges — and the canvas provide particular menus, that contain commands related to them, which are evoked through right mouse button clicks on these elements. These so called context menus are shown in Figure 3.2. Information is entered in a pop-up dialogue window with two input fields "Path" and "Label" (Figure 3.3).

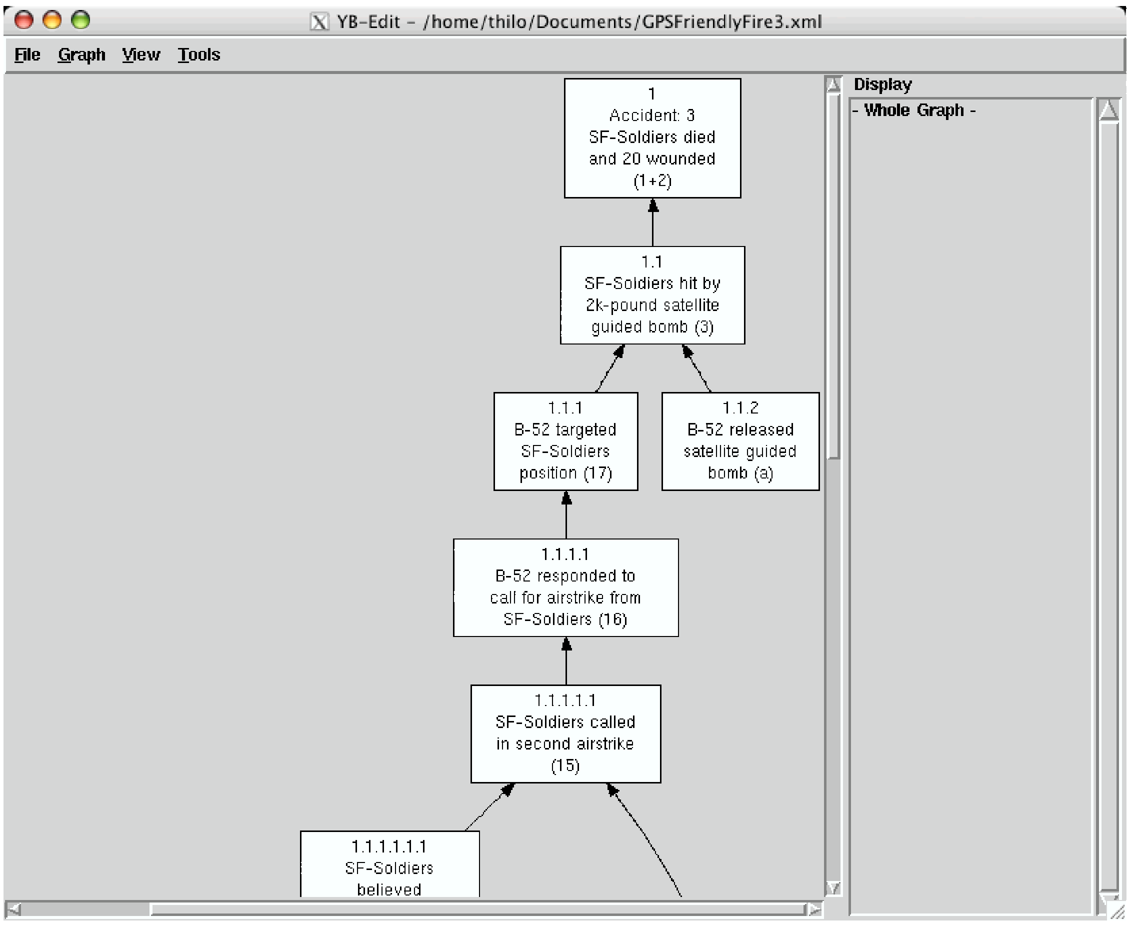

YB-Edit - /home/thilo/Documents/GPSFriendlyFire3.xml
File Graph View Tools
1 Accident: 3 SF-Soldiers died and 20 wounded (1+2)
1.1 SF-Soldiers hit by 2k-pound satellite guided bomb (3)
1.1.1 B-52 targeted SF-Soldiers position (17)
1.1.2 B-52 released satellite guided bomb (a)
1.1.1.1 B-52 responded to call for airstrike from SF-Soldiers (16)
1.1.1.1.1 SF-Soldiers called in second airstrike (15)
1.1.1.1.1.1 SF-Soldiers believed
Display
- Whole Graph -

Figure 3.1.: The program window of YB-Edit. The "Friendly Fire" accident WB Graph is shown on the canvas.

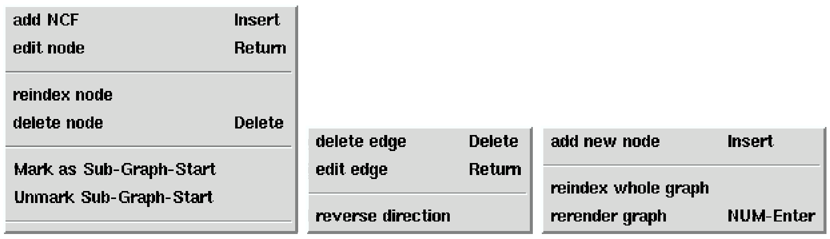


Figure 3.2.: The context pop-up menus of YB-Edit. From left to right: node menu, edge menu, and canvas menu.

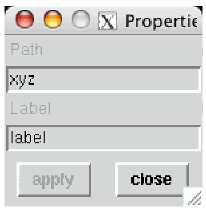


Figure 3.3.: The pop-up dialogue window of YB-Edit.

3.4. GOMSL Analysis of YB-Edit Use

3.4.1. The Procedure

GOMSL models consist of descriptions of the device, the task, and the methods and rules to accomplish the task with the specified device.

For a GOMSL analysis, a specific task is needed that can be fed to the production system. To obtain a realistic task I observed the development of the WB Graph of the "Friendly Fire" accident (see section 1.1.3) by RVS group members. In this way I received a sequence of subtasks — each one corresponding to one of the high level goals determined in section 2.3[1], that, when carried out, result in the WB Graph in Figure 1.2.

The definition of the device is done by a visual description that represents the GUI components of YB-Edit and the "Friendly Fire" WB Graph.

The methods defining the production system are those determined in section 2.3. The sub methods were obtained through the same observation I used to specify the task, and through exhaustive examination of YB-Edit itself to detect details such as keyboard input foci in particular states of the interface elements.

The selection rules, which form the other part of the production system, could be derived from the high level goals. Higher cognitive tasks such as determining

[1]When I speak of high level goals in relation with GOMS, I am always referring to the high level goals identified in section 2.3.

the place where to enter a node are not part of the model, since GOMS can not handle such tasks and they are independent of the GUI used.

3.4.1.1. Dealing with related tasks

Some observed actions could not be referred to the identified high level goals and are not part of the use of YB-Edit in direct. They arise from interaction with the working environment of the computer to deal with the List of Facts. These were conspicuous changes from YB-Edit to a text editor and back, that conduced to look up the next fact to add to the graph or to add new facts to the List of Facts. Since these tasks obviously belonged to the use of YB-Edit, I integrated them in the model and added a minimalistic text editor GUI representation to the device description.

Another action that was not determinable by the HTA is a feature of YB-Edit: the reindex function. To make the nodes referable by the analyst, the nodes are provided with an index number. If the WB-graph is changed, it can happen that the index numbers get mixed up in the area of the change. To deal with this issue I integrated the use of this feature in the model.

3.4.2. The Model

I will only present examples of the GOMSL model of the YB-Edit use in this section. For the complete model please have a look at Appendix B.

3.4.2.1. The Tasks

The single tasks are represented through task instances. They consist of an identifier, a list of properties, and a reference to the following task. The type of a task is one of the high level goals or the related tasks. The first task in building the "Friendly Fire" WB Graph is to determine a fact[2] — the mishap:

```
Task_item: Task1
  Name is Task1.
  Type is determine_fact.
  Next is Task2.
```

Later, when the methods are discussed, it will become evident why it contains hardly any information (section 3.4.2.4). The second task is more telling. It defines the task of adding a new node to the presently empty WB Graph. Its further properties are the information the analyst would have gathered from the List of Facts:

[2]Determine fact is not to be mixed up with the determine necessary causal factor sub operation in the HTA in section 2 here. In the context of the GOMS analysis it stands for finding a fact in the List of Facts and is — perhaps — inauspiciously chosen.

```
Task_item: Task2
  Name is Task2.
  Type is add_new_node.
  Label is "Accident:␣3␣SF-Soldiers␣died␣and␣20␣wounded".
  Node_number is "1".
  Reference is "(1+2)".
  Next is Task3.
```

The tasks that represent the addition of necessary causal factors contain one more property. The node which represents the fact that was caused by the necessary causal factor must be specified:

```
Task_item: Task4
  Name is Task4.
  Type is add_ncf.
  Label is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".
  Reference is "(3)".
  Parent is "Accident:␣3␣SF-Soldiers␣died␣and␣20␣wounded".
  Next is Task5.
```

Other tasks are modelled in a similar way, and since the "Friendly Fire" WB Graph was built up almost straight forward, the shown tasks represent closely the rest of the task list.

3.4.2.2. The Device

The Graphical User Interface The Device is represented with the help of visual objects. Like task instances, they consist of an identifier and a list of properties. These properties are visual properties the user of the device sees, if he is looking at the object. YB-Edit's program window is represented this way:

```
Visual_object: Ybedit_program_window
  Type is window.
  Label is "YB-Edit".
```

A menu entry is represented through:

```
Visual_object: Add_new_node
  Type is menu_entry.
  Label is "add␣new␣node".
```

Other interface elements are represented in a similar way.

The WB Graph The WB Graph does belong to the device description, too, because it is displayed by YB-Edit. Since one can not alter the visual description from within a running GOMSL model, the whole graph must be defined beforehand and the GOMSL analyst must take care that the modelled user does not look at things, which are not existent at a given time. The WB-graph is represented through nodes:

```
Visual_object: Node_11
  Type is node.
  Label is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".
  Reference is "(3)".
```

```
    Children is "2".
    Child1 is "B−52␣targeted␣SF−Soldiers␣position".
    Child2 is "B−52␣released␣satellite␣guided␣bomb".
    Parents is "1".
    Parent1 is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
```

And edges:

```
  Visual_object: Edge_11−1
    Type is edge.
    Source is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
    Target is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
```

3.4.2.3. The Selection Rules

The selection rules are given in the perform graph actions method and determine which method is executed depending on the actual task in the form of if. . . then rules. In the model of YB-Edit use there is one rule for every high level task, rules to deal with the List of Facts, and one rule for re-indexing the graph. Every information needed to accomplish the task that can not be determined in the model (e.g. by looking at a visual object) are passed as working memory tags to the methods.

3.4.2.4. The Methods

The methods describe what a user of YB-Edit does to achieve a goal. YB-Edit provides contextual menus as interface to its functions to add a node or a necessary causal factor to the graph, and to remove a node and remove an edge from the graph. Adding an edge is done by drag and drop operation. If input of data is reasonable after a command, an input window pops up, where the data can be entered. The method for the add necessary causal factor goal, for example, therefore looks like this:

```
  Method_for_goal: Add ncf using
                   <ncf_parent>, and <ncf_label>,
                   and <ncf_reference>, and <ncf_node_number>
    Step 1. Accomplish_goal: Issue context_command using
                             "add␣ncf", and node,
                             and <ncf_parent>, and nil.
    Step 2. Accomplish_goal: Enter ncf_description using
                             <ncf_label>, and <ncf_reference>,
                             and <ncf_node_number>.
    Step 3. Return_with_goal_accomplished.
```

In the method header, the working memory tags, which the method demands, are listed. Because all working memory tags are global in scope, every method has working memory tags identifiers with a unique prefix to make them pseudo local. This is not ideal, because it raises working memory demand, but is the only way to not get cluttered in complex models [Kieras 99].

The first step in the method is the call of a sub method to accomplish the issue context command goal with the command name, the context type, and the context as parameters. The nil parameter is an empty parameter, which is needed, because the called method requires four parameters.

The second step calls the sub method to accomplish the enter ncf description goal. After having issued the context command to add a ncf by accomplishing the first step, one would now be presented with the parameter input window by YB-Edit. The enter ncf description method is therefore provided with the necessary parameters to make expedient input in this context.

The Input Dialogue Window of YB-Edit YB-Edit uses the same input dialogue window for all node parameter input, but there are subtle differences in the interaction depending on the goal whose accomplishment evoked the window. These differences express in alternating input foci, changing preselection in the input fields, and different commands to evoke the dialogue window.

Due to complexity of the required modelling and the recommended number of steps per method [Kieras 99], there are now four different methods at the same hierarchy level, that describe the different interactions with the input window.

Handling Operations with the List of Facts The List of Facts is displayed in an external text editor, and the analyst switches back and forth between YB-Edit and the text editor to determine facts or to alter the List of Facts. Remarkably, none of the analyst used the system pasteboard to transmit information.

The method to determine a fact, for example, is represented by two program changes and a dummy method:

```
Method_for_goal: Determine fact
  Step 1. Accomplish_goal: Change_to program using
                           "Editor".
  Step 2. Verify "Find␣the␣fact␣that␣is␣causal␣factor".
  Step 3. Accomplish_goal: Change_to program using
                           "YB-Edit".
  Step 4. Return_with_goal_accomplished.
```

The first step in this method calls a sub method that switches the working environment to the text editor. The change to program method assumes the user to point to the editor window and click it. Using the keyboard to switch to the editor would consume almost the same time.

The dummy operator verify consumes a static amount of time. This is a mental operator provided by GOMSL to indicate a complex psychological process not further specified. In the determine fact method it is a placeholder for the identification process of a causal factor.

Step three changes the working environment back to YB-Edit.

Removing a Node To remove a node the analyst is required to detach its child nodes first. After having done so, he can call the context menu command to delete the node:

```
Method_for_goal: Remove node using <rm_label>
  Step 1. Look_for_object_whose Label is <rm_label> and_store_under <target>.
  Step 2. Decide:
          If Children of <target> is nil, Then
            Goto 7;
          If Children of <target> is "1", Then
            Goto 6;
          If Children of <target> is "2", Then
            Goto 5;
          If Children of <target> is "3", Then
            Goto 4.
  Step 3. Accomplish_goal: Remove edge using
                           Child4 of <target>, and <rm_label>.
  Step 4. Accomplish_goal: Remove edge using
                           Child3 of <target>, and <rm_label>.
  Step 5. Accomplish_goal: Remove edge using
                           Child2 of <target>, and <rm_label>.
  Step 6. Accomplish_goal: Remove edge using
                           Child1 of <target>, and <rm_label>.
  Step 7. Accomplish_goal: Issue context_command using
                           "delete_node", and <rm_label>, and nil, and nil.
  Step 8. Return_with_goal_accomplished.
```

First the visual object with the label of the node to delete is looked for.

In the second step, the number of child nodes is determined — an attribute of the visual object — and then the edges are removed by calling the remove edge method in steps three to six. The remove edge method consists of the execution of a context command.

In step seven the issue context command method is called to delete the node.

Inserting a node To insert a node between two nodes with a causal relation, requires the analyst to restructure the graph by hand:

```
Method_for_goal: Insert node using
                 <in_parent>, and <in_label>,
                 and <in_reference>, and <in_child1>,
                 and <in_child2>, and <in_child3>,
                 and <in_child4>
  Step 1. Accomplish_goal: Add ncf using
                           <in_parent>, and <in_label>,
                           and <in_reference>, and nil.
  Step 2. Accomplish_goal: Change parent using
                           <in_parent>, and <in_label>,
                           and <in_child1>, and <in_child2>,
                           and <in_child3>, and <in_child4>.
  Step 3. Return_with_goal_accomplished.
```

First an ncf is added to the former parent node of the existing nodes.

In the second step, the parent node for the given child nodes are changed by calling the change parent method. The accomplishing of this method requires the removal of the edges between the former parent node and the child nodes, and the addition of the child nodes as necessary causal factors to the inserted node.

Counting In the creation of a WB Graph there are actions, such as merge nodes or split node, that require repetition depending on object properties such as the number of child nodes or parent nodes. Since GOMSL has no operator to count or loop, I hardcoded these functions to a degree of four nodes, what may better represents human cognition, but resulted in — often unused — extra steps, as can be seen in the remove node method.

Issuing a Command All commands except two — add an edge and change source node of an edge — are invoked by selecting a contextual menu entry. The analyst looks for the object representing the context, (right-) clicks it, is then presented a menu, looks for the adequate menu entry, and clicks it. Browsing the menu is possible, too, but is not modelled because it nearly consumes the same amount of time. This is the method:

```
Method_for_goal: Issue context_command using
                 <icc_command>, and <icc_context>,
                 and <icc_label1>, and <icc_label2>
  Step 1. Decide:
          If <icc_context> is canvas, Then
            Accomplish_goal: Click_at item using
                             nil, and <icc_context>;
          If <icc_context> is edge, Then
            Accomplish_goal: Click_at edge using
                             <icc_label1>, and <icc_label2>;
          If <icc_context> is node, Then
            Accomplish_goal: Click_at item using
                             <icc_label1>, and <icc_context>.
  Step 2. Accomplish_goal: Click_at item using
                           <icc_command>, and menu_entry.
  Step 3. Return_with_goal_accomplished.
```

First, the context type is determined to accomplish the corresponding method to click at the object. The click at methods include the looking for an object of a specific type equipped with specific type dependent attributes.

The next step invokes the method to accomplish clicking at a menu item with a specific label.

3.5. The Results

3.5.1. Modelling with GOMSL

GOMSL offers no possibility to formulate counting or loops, what sometimes complicated the modelling. A facility to formulate repetitive tasks that regards human cognition would help a lot to formulate complex tasks like those occurring during a WB-Analysis.

The size and position of the visual objects can not be specified in GOMSL, which uses a variant of Fitt's Law. Pointing to an object therefore consumes a static amount of time, what may have negative influence on pop-up menu navigation time estimates.

The representation of the device does not change according to what is happening by task execution. The analyst must provide all states of the device that occur during accomplishing the goals in advance. To make sure that nothing wrong is detected by the simulated user, the objects often must be detected by more attributes than in reality.

When models of alternate designs are made, the similar workarounds must be used to ensure the comparability.

3.5.2. Using GLEAN3

When the model is processed with GLEAN3, it must first be loaded and then compiled. If the model compiles, that does not necessarily mean that the model is error free. To further debug the model, the compiled model is executed and the log is thoroughly checked for inconsistencies, such as visual objects not found. The learning time analysis can be done before the model is executed, but it is safer to debug it first.

3.5.3. Learning Analysis

Figure 3.4 displays the results of the learning analysis. The model consists of 34 different methods with a total of 216 steps. Although I shared as many sub methods as possible, there are a lot of specialised methods with many steps. Only 6 of the 216 steps — 2.78 percent — are learnable.

As mentioned earlier, there are four different methods to deal with the node or necessary causal factor description. The methods are so different from each other, that nothing can be learned from the other methods.

The remove method is — apart from the selection rules called perform graph actions — the method with the most steps, because removing a node in YB-Edit requires manually removing all sub nodes.

The methods change parent, change child, add as parent, and add as child are utility methods of the merge nodes, split node, and insert node methods and basically deal with the inability of GOMSL to count or loop. The number of steps is not dependent of the interface of YB-Edit.

Method Units:	number	learned	to learn	Source Method
Add edge	3	0	3	
Add fact	7	0	7	
Add ncf	4	0	4	
Add new_node	4	0	4	
Add_as child	10	2	8	Add_as parent
Add_as parent	10	0	10	
Alter edge	7	0	7	
Alter node_description	7	0	7	
Change child	10	2	8	Change parent
Change parent	10	0	10	
Change_to program	3	0	3	
Click_at edge	5	0	5	
Click_at item	5	0	5	
Create wb_graph	7	0	7	
Determine fact	5	0	5	
Double_click_at item	5	0	5	
Drag edge	8	0	8	
Drag item	8	0	8	
Enter fact_description	4	2	2	Enter node_number
Enter index_number	4	0	4	
Enter ncf_description	9	0	9	
Enter new_node_description	8	0	8	
Enter node_number	3	0	3	
Enter node_text	3	0	3	
Extend node_description	6	0	6	
Insert node	4	0	4	
Issue context_command	6	0	6	
Merge nodes	6	0	6	
Perform graph_actions	18	0	18	
Reindex graph	3	0	3	
Reindex node	3	0	3	
Remove edge	3	0	3	
Remove node	12	0	12	
Split node	6	0	6	
Totals:	216	6	210	

Figure 3.4.: Learning analysis of YB-Edit use.

3.5.4. Execution Times

The execution times are displayed in Figure 3.5. The table contains the number of times each method was executed, the total time spent in a method, the average time spent executing a method, and the percent of total execution time spent in a method. The percentages do not sum up to 100 percent because the methods are hierarchically structured.

Method for goal	Freq.	Subtotal	Avg. Time	% of Total
Add edge	2	11.5	5.75	1.63
Add fact	3	70.8	23.6	10.04
Add ncf	18	385.45	21.414	54.68
Add new_node	1	24.65	24.65	3.5
Alter edge	1	11.7	11.7	1.66
Change parent	1	11.85	11.85	1.68
Change_to program	38	117.1	3.082	16.61
Click_at edge	1	2.75	2.75	0.39
Click_at item	86	245.7	2.857	34.85
Create wb_graph	1	704.95	704.95	100
Determine fact	16	120	7.5	17.02
Double_click_at item	1	3.35	3.35	0.48
Drag item	2	11.3	5.65	1.6
Enter fact_description	3	35.55	11.85	5.04
Enter index_number	3	6.6	2.2	0.94
Enter ncf_description	18	280.65	15.592	39.81
Enter new_node_description	1	18.85	18.85	2.67
Enter node_number	1	0.8	0.8	0.11
Enter node_text	40	287.5	7.188	40.78
Extend node_description	2	20.6	10.3	2.92
Insert node	1	30.45	30.45	4.32
Issue context_command	22	124.7	5.668	17.69
Perform graph_actions	41	643.35	15.691	91.26
Remove edge	1	5.75	5.75	0.82

Figure 3.5.: Execution times of YB-Edit.

The creation of the WB Graph assuming an expert user would take 704.95 seconds with only a little cogitation and an instantaneous response by the system. The create wb_graph method is the top most method and hence represents the whole creation of the "Friendly Fire" WB Graph.

Apart from the selection rules in perform graph actions, the most time — 54.68 percent of total — was spent adding 18 necessary causal factors. The average execution time of the method is 21.41 seconds, what is almost the average execution time of adding a new node or adding a new fact. Most of the time these methods spend for entering the descriptions; remember that nobody used the system pasteboard. Entering text makes up 40.78 percent of the total execution time.

The program changes are noticeable, too. There are 38 of them, two for every node in the graph, consuming 117.1 seconds, what is 16.61 percent of the total execution time and 2.9 seconds less as it takes to determine a fact (120 seconds).

3.5.5. Visual Operators and Memory Demand

There is a number of 92 visual operations. If one bears in mind, that there is a list with around 20 facts to discover and graph with the same amount of nodes, this seems a lot to look at.

There were 81 working memory content replacements, and peak of four working memory contents stored at the same time.

3.5.6. Interaction with YB-Edit

3.5.6.1. Pop-up Menus

The pop-up menus are a form of indirect user engagement, which is not recommended for direct manipulation interfaces. They are only a clickable representation of typed commands, where communication to the system occurs through an invisible intermediary.

The issuing of a context command requires evoking a specialised method just to execute a command. Most of the time further interaction is required to accomplish a task.

3.5.6.2. Two Basic Interaction Forms

YB-Edit utilises two basic interaction forms: the earlier mentioned contextual pop-up menus and the drag and drop gesture, which is used for adding an edge to determine an existing node as necessary causal fact of another existing node.

Although it is more intuitive in its use, it adds a second specialised method that covers this gesture to the model.

3.5.6.3. The Input Dialogue Window

The input dialogue window is inconsistent in invocation and use. There are four different ways to let the window pop up, which control the state of the window. The window has four different modes without clear signs to detect them. It required a lot of GOMSL modelling work, which is an indication for need of optimisation [Card 83, Kieras 97a, Kieras 99].

3.5.6.4. The External List of Facts

A large amount of time is consumed for operations that have to do with the List of Facts. For every node that is added to the graph, the analyst switches over to the editor, determines a causal factor, keeps it in mind, switches back to YB-Edit, and after creating a node, reenters the information kept in mind. Only entering the text, the analyst has previously entered in a text editor, consumes around 40 percent of the whole time required to build a WB Graph, and the program switching consumes another 16 percent. The procedure requires the analyst to use his memory as a buffer for the description of the facts. This and the retyping are possible sources of error. Fixing this issue will halve the total execution time and eliminate sources of error.

3.5.6.5. Errors

A user error might occur when a causal factor is added to the graph, which is not present in the List of Facts at that time. After adding such factor to the graph, the analyst is required to add it to the List of Facts, which includes giving the fact an index number, and then adding this index number as reference to the corresponding node description. The latter can be missed easily, since the higher level goal — the addition of the node to the WB Graph — is already accomplished, and the analyst therefore stops accomplishing related sub tasks. In fact, this has happened during my observations. See node 1.1.1.1.1.1.2.2. in the "Friendly Fire" WB Graph in Figure 1.2 and the entry missing at the end of the corresponding List of Facts ("c. GPS coordinate representation") in Figure 1.1, which are unchanged results of the observed analysis.

The analysis of YB-Edit use unveiled another possibility for user error. It occurs when inserting a node between two nodes with a causal relation. After having inserted the node and redirected the edge, the main goal is reached, but WB-Edit requires the analyst to call the reindex function now, since the node index numbers got mixed up. See node 1.1.1.1.1.1.4.2 and node 1.1.1.1.1.1.4.1.1 in Figure 1.2. Node 1.1.1.1.1.1.4.2 was added as a sibling node to node and hence became its actual number. Afterwards, the edge between node 1.1.1.1.1.1.4.1.1 was changed to point to node 1.1.1.1.1.1.4.2. After a re-indexing of the graph the nodes would have obtained their correct index, but this was obviously forgotten.

This kind of errors is referred to as problem of closure [Dix 98].

3.5.6.6. Removing a Node

The removal of a node that is not a leaf requires the detaching of the child nodes by the user. This is a time consuming routine task which requires no decision taking by the user.

3.5.6.7. Insertion of a Node

As the removal of a node, inserting a node between two existing nodes with a causal relation requires a lot of manual operation, although it is a routine task, when the two existing nodes are specified. As seen in section 3.5.6.5, this method is a possible cause of error.

3.5.6.8. Single Object Manipulation

The context pop-up menus are evoked by right-clicking a screen object. This can be the whole WB Graph represented by the canvas, a node, or an edge. Because combination of several objects by selection is not possible, only attributes of a single object can be manipulated.

4. The New User Interface Design

4.1. Basic Concepts

4.1.1. Direct Manipulation

There must be only one basic interaction gesture for the manipulation of the WB Graph to make the interface learnable with ease. This must be a gesture that allows fast direct manipulation with user engagement. The drag and drop method used by YB-Edit to establish causal relations is a very intuitive and fast mouse interaction form, because it carries all necessary information in two pointing operations, that correspond to the natural locus of interest of the user, and one press and release of the mouse button.

To allow for interaction with multiple objects, a method to select several single screen objects, even if they are non-adjacent is required.

4.1.2. Managing the Attributes

The textual input of node attributes should be consistent. Because there are only few attributes that are set in the dialogue, the information of a selected node can be displayed all the time without cluttering the screen. This allows for a uniform input focus placement and preselection of text data. Furthermore the user is provided all information in the same interaction frame and is supported in familiarising with the user interface, since its layout does not change.

If zooming out the graph to gain overview, this is a possibility to gather node detail information without having to zoom in again. It may be useful to display further attributes of a node, such as its necessary causal factors or its effects, to give more details and to allow for further textual input.

4.1.3. Incorporation of the List of Facts

Like Bernd Sieker encourages in [Sieker 04], the List of Facts should be incorporated by YB-Edit.

There must be at least the possibility to load, display, modify and save the List of Facts with YB-Edit. The List of Facts contains the base information for creating a WB Graph. The Analyst is required to work with these information throughout the whole creation of the graph and is sometimes even required to make changes to it (see section 2.2). This does not necessarily mean that the List of Facts must be created using YB-Edit, because there exist many tools that are good at creating and dealing with indexed lists, but by integrating the List of Facts all process relevant information is in one place and can be used as source for drag and drop gestures.

Integrating the List of Facts bears several great advantages. A fact can be determined without context switching to another program. The information is directly usable without utilising working memory and the time consuming re-entering of previously entered text. The facts can be added to the WB Graph using drag and drop operations. Potentially required changes to the node description can be made afterwards. Furthermore the reference of a node to a fact can be made automatically.

In a similar way, a missing fact can be added to the List of Facts without leaving the program. If the analyst creates a node without corresponding fact, the node description can be used to generate an entry in the List of Facts which can be edited later. This is a way to gather a List of Facts, if the Why-Because List was used at first to arrange the facts.

By integrating the List of Facts the user errors observed in YB-Edit's use do not occur any more.

It can be useful to highlight the referenced facts of a selected node in the List of Facts, or to mark facts that are already used in the graph, but this will not be discussed here.

4.1.4. Automation

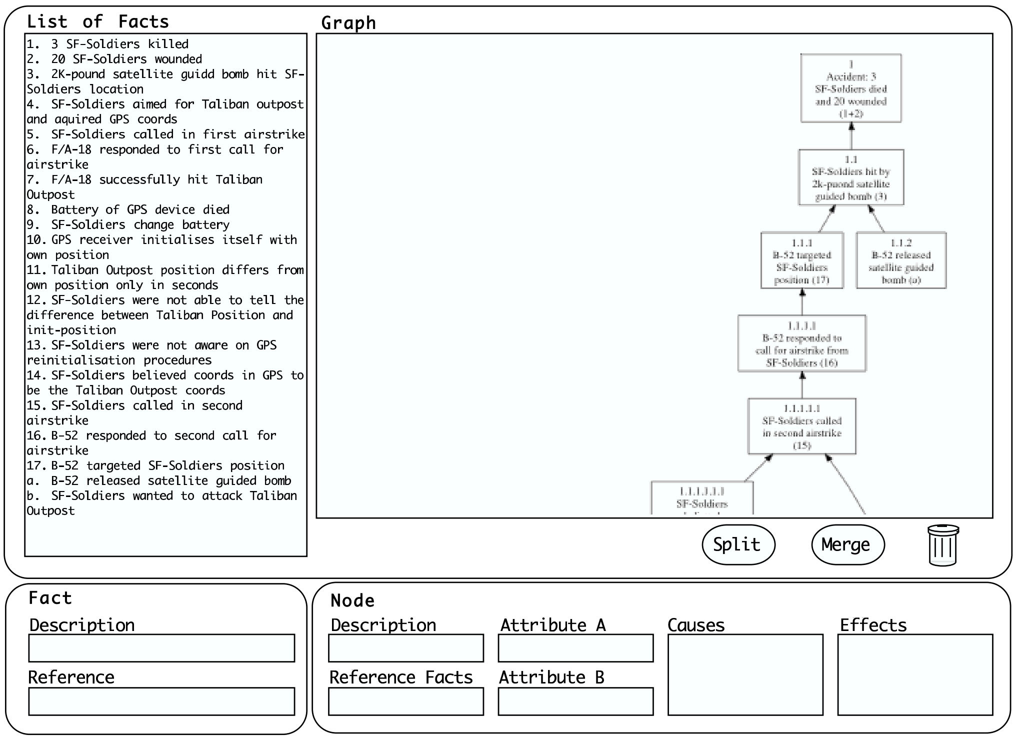
The indexing of the graph should happen without the user initiating it. The WB Graph nodes are indexed depending on their hierarchy level. Since user interaction is not required the operation that can be automated completely.

The removal of the edges pointing from child nodes to a node that is to be deleted and the routine tasks required to insert a node should be automated. No decision by the user is required, so there is no need to burden him with this work.

4.2. The Design

4.2.1. A GUI Sketch

Figure 4.1 shows a sketch of a new GUI design that supports the concepts stated in section 4.1. Please note, that it is not meant to be a suggestion for the graphic design of the interface.


List of Facts

1. 3 SF-Soldiers killed
2. 20 SF-Soldiers wounded
3. 2K-pound satellite guidd bomb hit SF-Soldiers location
4. SF-Soldiers aimed for Taliban outpost and aquired GPS coords
5. SF-Soldiers called in first airstrike
6. F/A-18 responded to first call for airstrike
7. F/A-18 successfully hit Taliban Outpost
8. Battery of GPS device died
9. SF-Soldiers change battery
10. GPS receiver initialises itself with own position
11. Taliban Outpost position differs from own position only in seconds
12. SF-Soldiers were not able to tell the difference between Taliban Position and init-position
13. SF-Soldiers were not aware on GPS reinitialisation procedures
14. SF-Soldiers believed coords in GPS to be the Taliban Outpost coords
15. SF-Soldiers called in second airstrike
16. B-52 responded to second call for airstrike
17. B-52 targeted SF-Soldiers position

a. B-52 released satellite guided bomb
b. SF-Soldiers wanted to attack Taliban Outpost

Graph
1 Accident: 3 SF-Soldiers died and 20 wounded (1+2)
1.1 SF-Soldiers hit by 2k-puond satellite guided bomb (3)
1.1.1 B-52 targeted SF-Soldiers position (17)
1.1.2 B-52 released satellite guided bomb (a)
1.1.1.1 B-52 responded to call for airstrike from SF-Soldiers (16)
1.1.1.1.1 SF-Soldiers called in second airstrike (15)
1.1.1.1.1.1 SF-Soldiers
Split
Merge
Fact
Description
Reference
Node
Description
Attribute A
Causes
Effects
Reference Facts
Attribute B

Figure 4.1.: Sketch of a new GUI design

The main part of the GUI consists of an area where the List of Facts is displayed on the left and the WB Graph drawing canvas on the right. Direct below the graph drawing canvas there are three panels, to which objects can be dragged. They are: a split panel, a merge panel, and a trash can.

Below are two dialogues that display the attributes of a selected fact, or node, respectively. There are two separate dialogues to avoid different modes of operation of the interface elements.

Selection of single objects — facts, nodes, and edges — is done by clicking. Multiple objects can be selected via rubber band selection or by shift clicking, if the nodes are non-adjacent. A single click on the graph canvas or the List of Facts area deselects anything and empties the dialogues. The keyboard focus is the description field of the node or fact dialogue, corresponding to what was clicked.

If, for example, the user selects node 1, the node attribute dialogue displays the description "Accident: 3 SF-Soldiers died and 20 wounded", the referenced facts "1, 2", and the cause "Soldiers hit by 2k-pound satellite guided bomb". The description field is preselected. To protect the user's work[1], the preselected text must be explicitly erased by hitting the delete key. Any other key will deselect the text and place the insertion mark at the end of the text. This adds only one keyboard hit to the interaction, but there is no risk of accidentally erasing valuable information.

The position and size of the elements is arbitrary at this time, since they can not defined in GOMSL and hence have no influence on the results.

4.2.2. The Interaction

The high level goals determined in section 2.3, the tasks concerning the List of Facts stated in section 3.4.1.1, and the concepts developed in section 4.1 form the basis for the interaction design.

4.2.2.1. Accomplishing of the High Level Goals

Add a Node There are two ways to add a node or a top node to the graph, depending on whether a corresponding fact is in the List of Facts or not. In the first case, the fact is dragged to the canvas and dropped. In the latter, the graph canvas is clicked, to deselect anything and place the text cursor in the node description field. Then the text is entered using the keyboard. After hitting the return key the new node is placed on the canvas, and a corresponding fact entry in the List of Facts is created. The new node is the active selection.

Add a Necessary Causal Factor To add a necessary causal factor to the graph, the corresponding fact is dragged from the list of facts to the chosen parent node. The new node is the active selection.

A second possibility to add a necessary causal factor is to add a new node to the graph first, and add an edge between the new node and the chosen parent node then. This method is a method to chose, if there is no corresponding fact in the List of Facts.

[1]To protect the user's work was one basic principle of the Macintosh Project, see [Raskin 00] for some information.

Insert a Node To add a node between two nodes that are connected by an edge, the corresponding fact is dragged to the edge between the nodes and dropped. The so inserted node is the active selection.

Again, there is the possibility to first create a node, and then drag it to the edge.

Remove a Node To delete a node it is dragged to the trash can.

Change an Edge To change the source or target of an edge, the head or the tail of the edge is dragged to the desired node and dropped. To create a new edge between to nodes, the node that represents the necessary causal factor is dragged to the node that represents the effect and dropped. Edges are deleted by dragging them to the trash can.

Change a Node Description To change a node description, the node is selected at first. Depending on whether a completely new description should be entered, or an existing description should be complemented, the delete key is hit before typing text. When finished, the return key is hit. The node is still selected and the whole text is preselected.

Merge Nodes To merge nodes, they are selected, dragged to the merge panel, and dropped. The result node is the active selection and the keyboard focus is in the node description field. The nodes description is entered by typing the text and hitting the return key. The edges and references are carried over to the result node. If this is not the desired state, this must be changed by hand.

Split Nodes To split a node, it is dragged to the split panel and dropped. A new node with the same parent node as the original node appears. The new node is the active selection and the keyboard focus is the node description field. The node's description is entered by typing the text and hitting the return key.

This is a horizontal split. A vertical split of could be realised in the same way, but is already covered by the method to insert a node.

Operations with the List of Facts

Determining a Fact Look at the List of Facts area.

Adding a Fact At first, by clicking on an empty part of the List of Facts area, any selected object is deselected and the the keyboard input focus is placed in the fact description field. The text is entered by typing it and hitting the return key when having finished. A new fact entry is created, which is the active selection. The keyboard input focus is still the fact description field, but now the previously entered text is preselected.

Removing a Fact To remove a fact from the List of Facts it is dragged to the trash can.

4.2.3. Comments

The new interface supports both the List of Facts approach, and the Why-Because List approach, while it is optimised for the first. When the Why-Because approach is used, the analyst receives a List of Facts with no extra costs.

The description of integration of the List of Facts is incomplete, and supports only the basic operations required for the accomplishing the creation of the "Friendly Fire" WB Graph.

4.3. GOMSL Analysis of the New Interface Design Use

4.3.1. The Model

The model introduced here represents the use of a YB-Edit that integrates parts of the new interaction design to accomplish the same task as in section 3.4. The task represents the actions required to create the "Friendly Fire" WB Graph in Figure 1.2. The device representation and the methods to accomplish the goals are adapted to represent the new user interface design. The complete model is listed in Appendix C.

4.3.1.1. The Tasks

The Tasks to accomplish are nearly the same as in section 3.4.2.1, except that tasks that deal with the references to the List of Facts and the re-indexing of the WB Graph are no longer required. Therefore there are only 39 instead of 41 tasks now.

4.3.1.2. The Device

The Graphical User Interface The graphical user interface is represented as described in section 4.2.1 by visual objects. In the main these are visual objects representing the attribute dialogue input fields like this:

```
Visual_object: Fact_description
  Type is input_field.
  Label is "Fact␣Description".
```

The split and the merge panel, and the trash can are modelled in a similar way:

```
Visual_object: Split
  Type is field.
  Label is "Split".
```

Since all the other commands are realised through drag and drop operations with elements that represent data, no further graphical user interface components are required.

The WB Graph The WB-graph description is the same as the one in the GOMSL model of YB-Edit. The split and the merge methods create a new node without description, which had to be added to the graph model:

```
Visual_object: New_node
  Type is node.
  Label is "New␣Node".
```

The label of the object is not empty, as it would be in reality, but “New Node”, to make sure it is detectable by GOMSL’s “look at” function that cognises visual objects.

The List of Facts The interface displays the List of Facts and the fact entries are source objects of the drag and drop gesture, so there must be visual objects to represent the single facts:

```
Visual_object: Fact11
  Type is fact.
  Label is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Number is "11".
```

4.3.1.3. The Selection Rules

The selection rules had to be modified, since the methods to achieve the high level goals have changed. E.g. to add a new node or to add a new necessary causal factor the same method is selected, and every time a label or description is changed the change attribute method is called. Furthermore there are more methods that deal with the integrated List of Facts and the rule for re-indexing the graph is removed, since this is not required anymore.

4.3.1.4. The Methods

Everything is drag and drop now. The method to add a node or necessary causal factor looks like this:

```
Method_for_goal: Add node using
                 <an_fact>, <an_ptype>, and <an_parent>
  Step 1. Accomplish_goal: Drag item using
                           fact, and <an_fact>, and nil,
                           and <an_ptype>, and <an_parent>, and nil.
  Step 2. Return_with_goal_accomplished.
```

In the first step a method is called that accomplishes dragging the fact determined in the previous task to a target . If the type of the target is the canvas, a new node is produced. If it is a node, the result is a necessary causal factor. The methods remove node, add edge, alter edge, remove node, and remove fact are accomplished in a very similar way.

Text Input If required, the textual input is accomplished with the change attribute method:

```
Method_for_goal: Change attribute using
                 <ct_type>,and <ct_attribute>, and <ct_a_or_x>,
                 and <ct_original>, and <ct_new>
  Step 1. Accomplish_goal: Click_at item using
                           <ct_type>, and <ct_original>, and nil.
  Step 2. Decide:
          If <ct_attribute> is_not "Description", Then
            Accomplish_goal: Click_at item using
                             input_field, and <ct_attribute>, and nil.
  Step 3. Decide:
          If <ct_a_or_x> is alter, Then
            Keystroke DEL.
  Step 4. Type_in <ct_new_label>.
  Step 5. Keystroke CR.
  Step 6. Return_with_goal_accomplished.
```

In the first step the target of the attribute change is selected by clicking on it. In the next step is decided, if the selection of an input field is required. If anything else than the description is to be changed, the corresponding input field is clicked. In step three, the delete key is typed, if the complete text is to be replaced. The text is typed in step four, follow by a return key press in step five.

There is textual input in other methods, too, e.g. in the add fact method:

```
Method_for_goal: Add fact using
                 <af_description>
  Step 1. Accomplish_goal: Click_at item using list, and "List␣of␣Facts", nil.
  Step 2. Type_in <af_description>.
  Step 3. Keystroke CR.
  Step 4. Return_with_goal_accomplished.
```

Since the interface is designed in such a way that there is no need to specify an input field in this situation, the text input is handled in two steps, which are included in place (steps 2. and 3.).

Handling Operations with the List of Facts Since the List of Facts is now displayed in the same program window as the rest, there is no need to change to another program window anymore. The determine fact method therefore consists of only two steps:

```
Method_for_goal: Determine fact
  Step 1. Verify "Find␣the␣fact␣that␣is␣causal␣factor".
  Step 2. Return_with_goal_accomplished.
```

Removing a Node The remove node method consists only of one step, because there is no menu navigation and detaching of child nodes necessary anymore:

```
Method_for_goal: Remove node using
                 <rm_label>
  Step 1. Accomplish_goal: Drag item using
                           node, and <rm_label>, and nil,
                           and trash, and "Trash", and nil.
  Step 2. Return_with_goal_accomplished.
```

4.4. Results

4.4.1. Learning Analysis

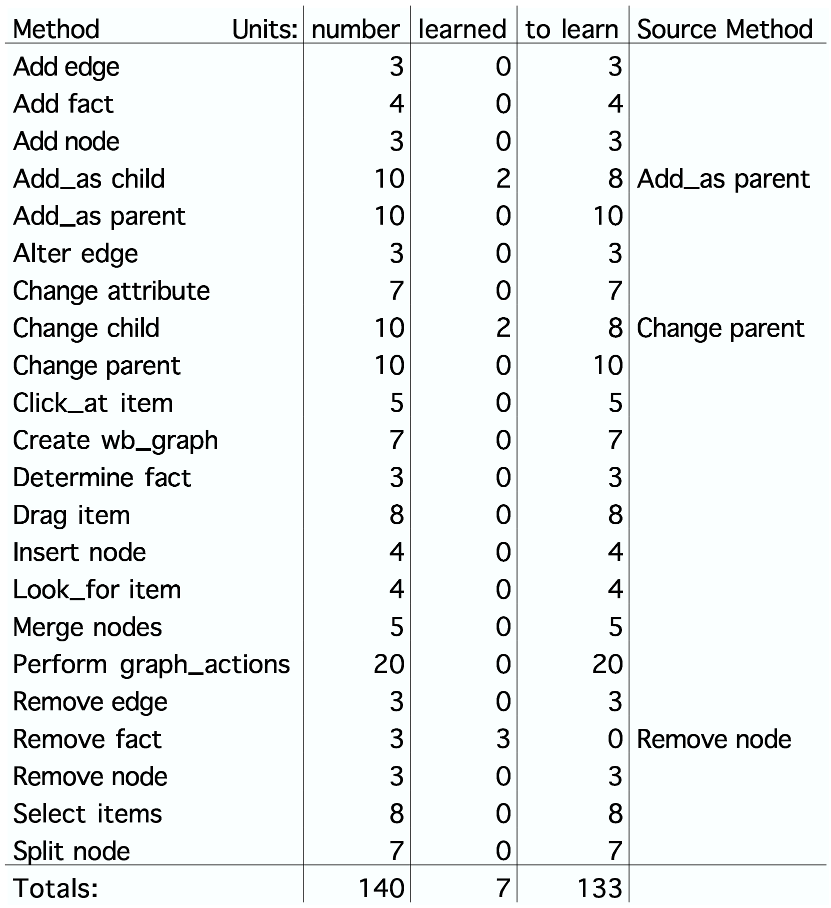
The results of the learning analysis are displayed in Figure 4.2. Because there are less specialised methods, the model consists of only 22 methods with a total of 140 steps. Half of the methods consist of four or less steps and just form a framework for a drag and drop operation. The reuse of the drag and drop method is not regarded by the learning analysis of GLEAN3, because of there are to much parameter that are different when it is called. Only the remove fact and the remove node have enough similarities that there is learnability. They consist solely of one drag and drop operation with difference in one parameter. Seven of the 140 steps can be learned from other methods. That is five percent of total.

As in the model of YB-Edit use, the methods change parent, change child, add parent, and add child are utility methods that deal with the inability of GOMSL to handle counting or loops.

4.4.2. Execution Times

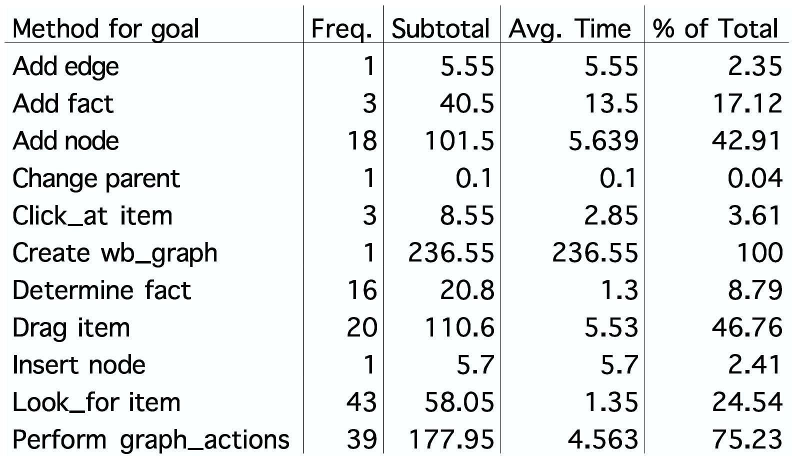
In Figure 4.3 the execution times are displayed. The creation of the WB Graph took now 236.55 seconds.

46.76 percent of the time, the user drags something. The average execution time of the drag item method is 5.53 seconds. Adding a node makes up 42.91



Method Units:	number	learned	to learn	Source Method
Add edge	3	0	3	
Add fact	4	0	4	
Add node	3	0	3	
Add_as child	10	2	8	Add_as parent
Add_as parent	10	0	10	
Alter edge	3	0	3	
Change attribute	7	0	7	
Change child	10	2	8	Change parent
Change parent	10	0	10	
Click_at item	5	0	5	
Create wb_graph	7	0	7	
Determine fact	3	0	3	
Drag item	8	0	8	
Insert node	4	0	4	
Look_for item	4	0	4	
Merge nodes	5	0	5	
Perform graph_actions	20	0	20	
Remove edge	3	0	3	
Remove fact	3	3	0	Remove node
Remove node	3	0	3	
Select items	8	0	8	
Split node	7	0	7	
Totals:	140	7	133	

Figure 4.2.: Learning analysis of the new interface design.



Method for goal	Freq.	Subtotal	Avg. Time	% of Total
Add edge	1	5.55	5.55	2.35
Add fact	3	40.5	13.5	17.12
Add node	18	101.5	5.639	42.91
Change parent	1	0.1	0.1	0.04
Click_at item	3	8.55	2.85	3.61
Create wb_graph	1	236.55	236.55	100
Determine fact	16	20.8	1.3	8.79
Drag item	20	110.6	5.53	46.76
Insert node	1	5.7	5.7	2.41
Look_for item	43	58.05	1.35	24.54
Perform graph_actions	39	177.95	4.563	75.23

Figure 4.3.: Execution times of the new interface design.

percent of the total time, and the average execution time of the method is with 5.64 seconds only 0.11 seconds higher as the drag items average execution time.

The three missing facts were added not by using the side effect of adding a node without corresponding fact, but by adding the facts first to the List of Facts, what takes 13.5 seconds on average. This is a long time compared to the other methods, what is caused by the typing of the description. Determining a fact now only takes 1.3 seconds on average.

The look for item method is a sub method of the drag item method. 24.54 percent of the whole execution time, the user is looking for what to drag and where to drag. The change parent method was called from the insert fact method, but had nothing to accomplish.

The perform graph actions method was called only 39 times. This means, there were only 39 tasks to accomplish.

4.4.3. Visual Operators and Memory Demand

The number of visual operations during the creation of the "Friendly Fire" WB Graph is 43. These are made up of 19 visual operations to look at the 19 facts in the List of Facts and one to look up the canvas to add the top node to the graph, 17 operations to look at nodes to add the necessary causal factors to the graph, and one to look at an edge to insert a node between two nodes. The remaining five visual operations are required to look at the source and target node for adding the edge representing a causal relation between them, and to look at the List of Facts for the deselection required ahead of the addition of the three missing facts.

There were 97 working memory content replacements, and peak of three working memory contents stored at the same time.

5. Comparison

5.1. Learnability

In the model of YB-Edit use the methods to achieve different high level goals are very specialised. Therefore a large amount of different sub methods is called by these high level methods. In contrast, the methods in the model of the new interface design use are mostly frameworks to provide a called drag and drop method with necessary parameters. In most cases this is enough to accomplish a high level goal. In some cases, such as adding a fact, providing additional information to the system is inevitable, what requires further operations to get executed. The amount of differing parameters that are passed to the drag item method from the high level methods is to high to satisfy the learnability definitions of GLEAN3 [Kieras 99]. The number of learnable steps is nearly as low as that of the model of YB-Edit use.

The model of YB-Edit use consists of 34 methods composed of 6.35 steps on average. Only 6 steps of the 216 steps representing the whole model are learnable. The new interface design use model consists of 22 methods with 6.36 steps on average, and 7 steps of the total 140 steps are learnable. If the methods required to formulate the framework of the models and interface independent tasks common to both models were not taken in account, the results would have become even more evident (see the tables in Figure 3.4 and 4.2).

5.2. Execution Times

The total execution time of new interface design use model is 2.98 times lower than that of the YB-Edit use model. Basically, this is due to the fact, that the user is not required to re-enter the fact descriptions anymore when creating a node. Therefore the average execution time of the add node method, for example, was shortened from 21.41 seconds to 5.82 seconds. The difference is 0.18 seconds more than the execution time of the enter ncf description method in the YB-Edit model.

The average execution time of determine fact method of the YB-Edit use model is 7.5 seconds. With the new design, the time is reduced to 1.3 seconds due to the fact that no more program changes are required any more.

The time to insert a node was reduced from 30.45 seconds in the YB-Edit use model to 5.7 seconds with the new interface design use model. This is due to the fact, that the new design provides a drag and drop operation for inserting a node between two nodes with a direct causal relation, while YB-Edit requires several other high level goal methods to be accomplished to get the task done.

On average the drag item method of the new interface design is only 0.138 seconds faster accomplished than the issue context command method.

To build the WB Graph with YB-Edit, 87 mouse click and 2 drag and drop operations are required. The creation of one node requires 4.53 mouse clicks on average. The new design required only three mouse clicks, but 20 drag and drop operations to build the WB Graph. This is 1.05 drag and drop operations per node.

The average execution times of the drag item and the click item method of new interface design use model are slightly faster than those in YB-Edit use model, what should not be the case.

5.3. The Models

In the new interface model most of the high level tasks are accomplished by evoking the drag item method. The mouse click method is required to deselect something, which only occurs if adding new facts or nodes to the List of Facts or WB Graph. When working with multiple objects at once, the click method is required to select non adjacent nodes.

The YB-Edit use model requires accomplishing the issue context command method with different context sensitive commands for resembling operations, and a drag and drop method to initiate the accomplishing of high level tasks. These methods can only be applied to one object at the same time.

With the new interface design attribute manipulation is uniformly handled via two dedicated dialogues, which are both always visible. The use of the time consuming type in operators was reduced to situations where completely new information must be entered. YB-Edit interface provides four slightly different pop up dialogue windows to change the node descriptions and therefore four different methods to deal with them. The List of Facts is handled by an external editor. This causes several specialised methods to change the nodes attributes, extra methods to deal with the editor, and often called methods for typing in text, that is already in digital form.

The automation of tasks that required no user decision simplifies the methods to remove a node, to insert a node, and to reindex the graph. The method for the latter is even non-existent in the new model. Inserting a node requires accomplishing the add ncf, add edge, and delete edge method if using YB-Edit. The new interface requires just accomplishing a drag item method to drag the fact to the edge between the nodes, which is a non-ambiguous operation which provides all necessary information to the program to change the data accordingly. The remove node method of the new interface requires no longer the deletion of the edges to sub nodes of the node to delete. This can save a lot of extra work, depending on the prior importance of the node.

The management of the references to the List of Facts entries presumed by the new interface design avoids the occurrence of the observed user errors. The reference is determined indirectly by the drag and drop method called by the add node and insert node method.

YB-Edit requires the user to look at a lot of things. For building the WB Graph consisting of 19 nodes 92 visual operations are required. The new interface design requires 43 visual operations, what is appropriate if considered that there were three facts to add to the List of Facts and one extra edge to insert into the Graph. The overhead of visual operations in YB-Edit is caused by twice calling the program change method per List of Facts operation, which includes one visual operator to look at the program window to click at, and the interaction overhead caused by the non-automation of some tasks.

The new design use model has with 97 working memory content replacements more different things to bear in mind over all than the model of YB-Edit use with 81 working memory replacements. The higher amount of memory replacements is caused by introducing the entries of the List of Facts as objects to act with, what causes more pseudoparameter passes per method in the model of the new design.

The peak of working memory contents, the user has to keep in mind is three with the new design, and four with YB-Edit.

5.4. Resume

The pure execution time for accomplishing the task of creating the "Friendly Fire" WB Graph is nearly three times lower with the new interface design.

The new interface design has one basic gesture which is sufficient to accomplish most of the high level goals. This way, the user is provided an easy to learn interface, which supports fast familiarisation.

The improvement through the drag and drop method is not a lower execution time of the method compared to the issue context command method in the model of YB-Edit use, but the effect for the straightforwardness of the interaction. The drag and drop gesture does not require the user to turn away from his natural locus of interest: when concentrated on the List of Facts to find a fact, the user can directly pick the one he finds and can directly search for the place to add it to the WB Graph with the fact at his hand. When the adequate node is found, the fact is simply dropped on it. This is done without having to explicitly evoke a command at first. An indicator for this is the reduced amount of visual operations. The integrated List of Facts supports the drag and drop gesture by providing the source objects. Changing back and forth to another program and remembering fact descriptions in not needed anymore.

The input dialogue windows in the new interface design are rarely required, but when, they behave in the same way in every situation. This expresses in the fact, that only one method was needed to specify the text input for the nodes and the facts. In some cases it is not even used, because the input interaction is reduced to pure typing of text without extra interaction. Therefore the accomplishing of task used for the analysis did not require the method to get accomplished once.

The two observed errors made during the creation of the "Friendly Fire" WB Graph can not be made with the new interface design anymore. The integration of the List of Facts and the automated indexing of the graph eliminated the problems of closure.

The new design requires the user to keep more different parameters for interaction in working memory, but only three at the same time.

6. Conclusion

6.1. Hierarchical Task Analysis

The Hierarchical Task Analysis very useful for gaining an overview of the procedure of a WBA and inevitable to determine the the high level goals for the analysis of YB-Edit use.

It requires some experience to carry out structured interviews. One tends to collect to much detail while acquiring high level tasks. The result is a flat list of sub operations, then. The record sheet I designed is very useful, although it provides to much space for one task description, and should be adapted to hinder the analyst from collecting too much detail on the same level. Very useful is the tool, that converts the textual representation of the hierarchy to a visual graph representation using a graph layout tool. This allowed me to concentrate on the subject matter and get visual representations in short time.

The whole diagram of performing a WBA is to large to include it in this theses — it is 2.8 meters wide, so I split it up into sub diagrams. Although the HTA does not go down to basic operations, the corresponding diagram became large very fast. This is caused by the characteristics of the presentation form, and the automatic layout, which tries to mimic the HTA diagram look by using a tree layout. To my knowledge, the fast growth of hierarchical task diagrams is not mentioned in literature.

I doubt that any process of doing a WBA is completely described by the diagrams in section 2.2. I only talked to a limited amount of interviewees, which all work in the same group, what sure had an influence on the outcomes of the analysis. Interviewing more Why-Because analysts will give a more complete picture. Nevertheless the results of the analysis were usable for the analysis of the use of YB-Edit.

6.2. GOMSL and GLEAN3

The GOMS Language was sufficient to model the complex task of using YB-Edit to create a complete WB Graph. In contrast to some other GOMS variants it is very well documented [Kieras 99]. With the detailed execution time, model and method complexity delivered by GLEAN3, predictions of the use of a planned or existent interface design can be made on the basis of the GOMSL model.

Like with HTA, the experience of the analyst influences the quality of the models and therefore the results of the analysis. Although the modelling process is described, there are no strict if. . . then rules that guide an analyst. The drag item and click at methods in the models of the use of YB-Edit and the new interface design should have the same execution times, but in fact have not. These slight average execution speed advantages in the order of magnitude of the tenth of a second might be due to a look at item method which the drag item and click item methods call. It was added to the new design's model to handle looking at different kinds of objects, what is required to deal with the included the List of Facts

Of most value is the development of the single methods to accomplish the goals. During the formulation of the smallest elements of an interaction even small inconsistencies stand out. The execution times and learning analysis delivered by GLEAN3 then provide the values that document the observations made during the modelling process.

I noticed four deficiencies while working with GOMSL and GLEAN3. First, there is no way to count and to formulate parameter dependent loops in GOMSL. This makes the formulation of some methods complicated. The remove node method in the model of the use of YB-Edit, for example, requires the accomplishing of a remove edge method for all the edges to the child nodes of the node to delete. In a higher programming language this is straightforward to formulate with some sort of loop statement, while GLEAN requires to explicitly formulate a rule for every possible case. I limited the amount of allowed child nodes and parent nodes to four (see Appendix B and Appendix C).

The second deficiency is the impossibility to specify size and position of visual objects in the device description and is commented in [Kieras 99]. On one hand, this makes the description of the device more easy, especially in cases where such informations are not available (e.g. a new design), but on the other hand the calculations done by GLEAN3 using Fitt's Law provide only flat rounding values. A good compromise might have been to use default values for the calculations in case none are specified in the visual object description. This would have made the comparison of the drag and drop and the contextual pop-up menu approach a little more realistic.

The third deficiency concerns the visual objects, too. There is no way to control from within the model, if a visual object is visible or not. This means that the methods that deal with visual objects must be designed in such a way, that they do not look at objects that are not present at the current state of the interface. This makes those methods more complicated and therefore unrealistic, and the visual objects need have more attributes than necessary. In the new interface design's model, for example, the drag and drop method called by the add node method found the node it should add to the graph instead of the fact with the same description and made it the object of the drag and drop operation. I had to specify the type of visual object to make the fact and the — in reality not yet present — node distinguishable in the model.

The last shortcoming I discovered is a problem with the so called pseudoparameters that hand through the necessary working memory contents to sub methods. These are all global "variables", what allowed some sub methods to overwrite working memory contents used by higher level methods, what had influence on the accomplishing of these higher level methods. I made the pseudoparameters local by using unique names for all pseudoparameters in every method. This raised the working memory demand of the model, but the memory model used by GLEAN3 is not meant to give a realistic picture of human brain functions anyway [Kieras 99] (see [Schulmeister 97, Kandel 00] for some details on brain functions or memory use). Since I have done so in both models, the comparability of models should be preserved.

6.3. YB-Edit Use

The analysis of YB-Edit use showed several areas of possible optimisation, which were used as basis for the new design. Furthermore the design of the new interface was directly based on the high level goals determined by the hierarchical task analysis of the process of doing a WBA, what guaranteed that the interface suits to the task.

The new interface design speeds the interaction up by factor of three, disburdens the user in several areas, and avoids two possible causes for problems of closure. This was achieved by extending the functionality of YB-Edit, optimising the use of gestures, and adapting the graphical user interface elements.

Since the analysis of YB-Edit use was designed to address the manipulation of a WB Graph, two aspects of the new interface design could not be further analysed: the manipulation of multiple objects at once and the manipulation of the List of Facts. There are some operations that could leave the List of Facts in an inconsistent state. E.g. the removing of a fact can cause a node to have no reference to the List of Facts any more, which is not desired.

6.4. Outlook

During its evolution CI-Edit, and later YB-Edit, was primarily optimised for functionality and stability, what is not rather astonishing when concerning the field of use of the program: system safety and security. Several interface inconsistencies probably have been caused by the integration of new required functions to the program. The process of doing a WBA was not analysed until now, what made it difficult to adapt a software tool to the process. Above all the library used in YB-Edit for displaying the graph is — while producing good looking graphs — limited to some basic interaction gestures.

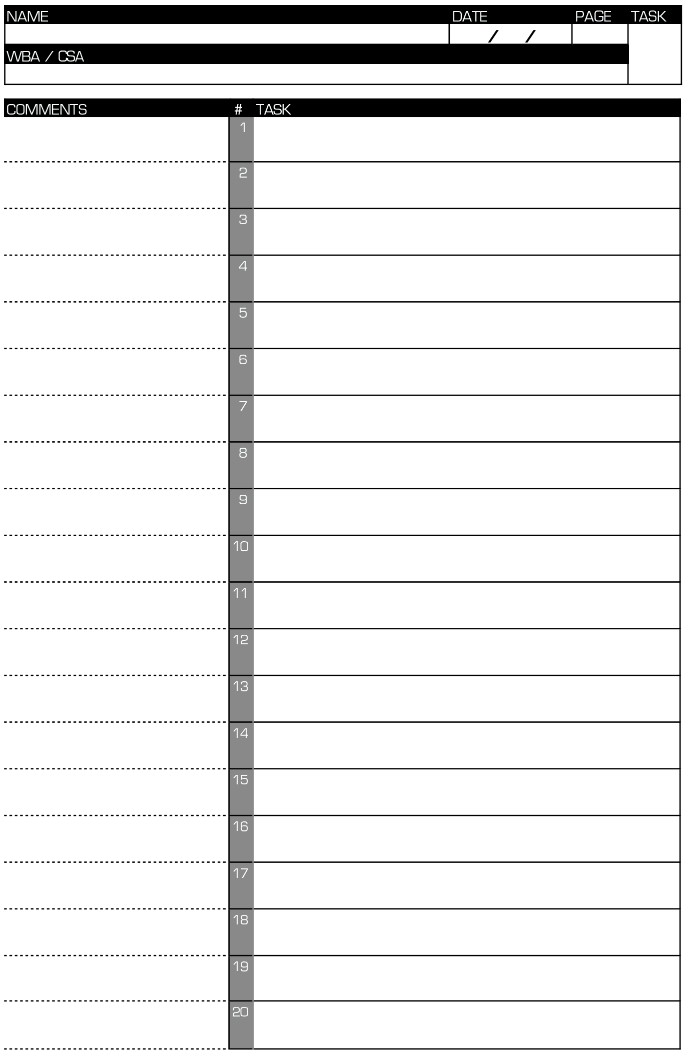
By integrating the new design in YB-Edit, the usability of the program will be greatly enhanced, and its use will become more safe. The advantages of the new interface design outweigh the work required to integrate the interface design. This is not a simple task, since not only a complete reimplementation of the graphical user interface is required, but also the extension of the functionality of YB-Edit.

Before this can be done, the manipulation of the List of Facts and the multiple object manipulation need further investigation and must be specified to make sure that they are no source of inconsistency or possible cause of error. To examine the methods of the new interface design not used in the "Friendly Fire" task, the creation of other WB Graphs can be observed to specify a realistic list of task instances. The existing models can be extended to make further investigation possible.

Interviews with external analysts that use WBA are needed to make sure that the process of doing a WBA is described completely in section 2.2.

The information obtained by the hierarchical task analysis of the WBA process can be used as basis for writing a documentation of performing a WBA. The models of YB-Edit and the new interface design use deliver valuable information for a user manual. In combination this yields the groundwork of a how-to manual for practical application of the WBA method.

A. HTA Record Sheet



NAME	DATE	PAGE	TASK
	/ /		
WBA / CSA			

COMMENTS	#	TASK
	1	
	2	
	3	
	4	
	5	
	6	
	7	
	8	
	9	
	10	
	11	
	12	
	13	
	14	
	15	
	16	
	17	
	18	
	19	
	20	

Figure A.1.: The HTA record sheet.

B. GOMSL Model of YB-Edit

```
Define_model: "yb−edit␣use␣GPS␣Friendly␣Fire"
  Starting_goal is Create wb_graph.

//
    ***************************************************************************

// Assumptions:
//   − a List of Facts exists
//   − yb−edit is allready running, presenting a new empty document
//   − an ordinary text editor is ready and displaying the List of Facts
//   − the canvas's size is big enough to display the whole graph :−)
//   − nodes have no more than four child and four parent nodes (this must also
//     true for the result nodes of merge operations!)
//
    ***************************************************************************


//
    ***************************************************************************

// The Tasks
// a list of task items
//
    ***************************************************************************


Task_item: Task1
  Name is Task1.
  Type is determine_fact.
  Next is Task2.

Task_item: Task2
  Name is Task2.
  Type is add_new_node.
  Label is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
  Node_number is "1".
  Reference is "(1+2)".
  Next is Task3.

Task_item: Task3
  Name is Task3.
  Type is determine_fact.
  Next is Task4.

Task_item: Task4
  Name is Task4.
  Type is add_ncf.
  Label is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Reference is "(3)".
  Parent is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
  Next is Task5.

Task_item: Task5
```

```
  Name is Task5.
  Type is determine_fact.
  Next is Task6.

Task_item: Task6
  Name is Task6.
  Type is add_ncf.
  Label is "B−52␣targeted␣SF−Soldiers␣position".
  Reference is "(17)".
  Parent is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Next is Task7.

Task_item: Task7
  Name is Task7.
  Type is determine_fact.
  Next is Task8.

Task_item: Task8
  Name is Task8.
  Type is add_ncf.
  Label is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Reference is "(16)".
  Parent is "B−52␣targeted␣SF−Soldiers␣position".
  Next is Task9.

Task_item: Task9
  Name is Task9.
  Type is determine_fact.
  Next is Task10.

Task_item: Task10
  Name is Task10.
  Type is add_ncf.
  Label is "SF−Soldiers␣called␣in␣second␣airstrike".
  Reference is "(15)".
  Parent is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Next is Task11.

Task_item: Task11
  Name is Task11.
  Type is determine_fact.
  Next is Task12.

Task_item: Task12
  Name is Task12.
  Type is add_ncf.
  Label is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Reference is "(14)".
  Parent is "SF−Soldiers␣called␣in␣second␣airstrike".
  Next is Task13.

Task_item: Task13
  Name is Task13.
  Type is determine_fact.
  Next is Task14.

Task_item: Task14
  Name is Task14.
  Type is add_ncf.
  Label is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Reference is "(13)".
  Parent is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Next is Task15.
```

```
Task_item: Task15
  Name is Task15.
  Type is determine_fact.
  Next is Task16.

Task_item: Task16
  Name is Task16.
  Type is add_ncf.
  Label is "SF-Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and␣
      Taliban␣outpost␣position".
  Reference is "(12)".
  Parent is "SF-Soldiers␣believed␣GPS-coords␣to␣be␣Taliban␣Outpost".
  Next is Task17.

Task_item: Task17
  Name is Task17.
  Type is determine_fact.
  Next is Task18.

Task_item: Task18
  Name is Task18.
  Type is add_ncf.
  Label is "Taliban␣Outpost␣position␣and␣SF-Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Reference is "(11)".
  Parent is "SF-Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Next is Task19.

Task_item: Task19
  Name is Task19.
  Type is determine_fact.
  Next is Task20.

Task_item: Task20
  Name is Task20.
  Type is add_ncf.
  Label is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Reference is "(10)".
  Parent is "SF-Soldiers␣believed␣GPS-coords␣to␣be␣Taliban␣Outpost".
  Next is Task21.

Task_item: Task21
  Name is Task21.
  Type is determine_fact.
  Next is Task22.

Task_item: Task22
  Name is Task22.
  Type is add_ncf.
  Label is "SF-Soldiers␣changed␣batteries".
  Reference is "(9)".
  Parent is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Next is Task23.

Task_item: Task23
  Name is Task23.
  Type is determine_fact.
  Next is Task24.

Task_item: Task24
  Name is Task24.
  Type is add_ncf.
  Label is "Battery␣of␣GPS␣receiver␣died".
  Reference is "(8)".
  Parent is "SF−Soldiers␣changed␣batteries".
  Next is Task25.

Task_item: Task25
  Name is Task25.
  Type is determine_fact.
  Next is Task26.

Task_item: Task26
  Name is Task26.
  Type is add_ncf.
  Label is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Reference is "(7)".
  Parent is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Next is Task27.

Task_item: Task27
  Name is Task27.
  Type is determine_fact.
  Next is Task28.

Task_item: Task28
  Name is Task28.
  Type is add_ncf.
  Label is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Reference is "(6)".
  Parent is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Next is Task29.

Task_item: Task29
  Name is Task29.
  Type is determine_fact.
  Next is Task30.

Task_item: Task30
  Name is Task30.
  Type is add_ncf.
  Label is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Reference is "(4)".
  Parent is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Next is Task31.

Task_item: Task31
  Name is Task31.
  Type is determine_fact.
  Next is Task32.

Task_item: Task32
  Name is Task32.
  Type is insert_ncf.
  Label is "SF−Soldiers␣called␣for␣first␣airstrike".
  Reference is "(5)".
  Parent is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Child1 is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Next is Task33.

// reindexing is required now, but was forgotten!
```

```
Task_item: Task33
  Name is Task33.
  Type is add_ncf.
  Label is "B−52␣released␣satellite␣guided␣bomb".
  Parent is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Next is Task34.

Task_item: Task34
  Name is Task34.
  Type is add_fact.
  Description is "B−52␣released␣satellite␣guided␣bomb".
  Next is Task35.

Task_item: Task35
  Name is Task35.
  Type is extend_node_description.
  Label is "B−52␣released␣satellite␣guided␣bomb".
  Extension is "(a)".
  Next is Task36.

Task_item: Task36
  Name is Task36.
  Type is add_ncf.
  Label is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Parent is "SF−Soldiers␣called␣in␣second␣airstrike".
  Next is Task37.

Task_item: Task37
  Name is Task37.
  Type is add_fact.
  Description is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Next is Task38.

Task_item: Task38
  Name is Task38.
  Type is extend_node_description.
  Label is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Extension is "(b)".
  Next is Task39.

Task_item: Task39
  Name is Task39.
  Type is add_edge.
  Source is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Target is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Next is Task40.

Task_item: Task40
  Name is Task40.
  Type is add_ncf.
  Label is "GPS␣coordinate␣representation".
  Parent is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Next is Task41.

Task_item: Task41
  Name is Task41.
  Type is add_fact.
  Description is "GPS␣coordinate␣representation".
  Next is None.

// adding the reference to the node description was forgotten
```

```
//
    ***********************************************************************
// Visual Objects
//
    ***********************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The Program Windows
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Ybedit_program_window
  Type is window.
  Label is "YB-Edit".

Visual_object: Editor_program_window
  Type is window.
  Label is "Editor".

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The Editors interface elements
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Insertion_point
  Type is insertion_point.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// YB-Edits interface elements
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Canvas
  Type is canvas.

Visual_object: Add_new_node
  Type is menu_entry.
  Label is "add␣new␣node".

Visual_object: Reindex_whole_graph
  Type is menu_entry.
  Label is "reindex␣whole␣graph".

Visual_object: Rerender_graph
  Type is menu_entry.
  Label is "rerender␣graph".

Visual_object: Delete_edge
  Type is menu_entry.
  Label is "delete␣edge".

Visual_object: Edit_edge
  Type is menu_entry.
  Label is "edit␣edge".

Visual_object: Reverse_direction
  Type is menu_entry.
  Label is "reverse␣direction".

Visual_object: Add_ncf
  Type is menu_entry.
  Label is "add␣ncf".

Visual_object: Edit_node
  Type is menu_entry.
  Label is "edit␣node".

Visual_object: Reindex_node
  Type is menu_entry.
  Label is "reindex␣node".

Visual_object: Delete_node
  Type is menu_entry.
  Label is "delete␣node".

Visual_object: Path_input_field
  Type is input_field.
  Label is "Path".

Visual_object: Label_input_field
  Type is input_field.
  Label is "Label".

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The nodes representing the graph
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Node_1
  Type is node.
  Label is "Accident:␣3␣SF-Soldiers␣died␣and␣20␣wounded".
  Reference is "(1+2)".
  Children is "1".
  Child1 is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".

Visual_object: Node_11
  Type is node.
  Label is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".
  Reference is "(3)".
  Children is "2".
  Child1 is "B-52␣targeted␣SF-Soldiers␣position".
  Child2 is "B-52␣released␣satellite␣guided␣bomb".
  Parents is "1".
  Parent1 is "Accident:␣3␣SF-Soldiers␣died␣and␣20␣wounded".

Visual_object: Node_111
  Type is node.
  Label is "B-52␣targeted␣SF-Soldiers␣position".
  Reference is "(17)".
  Children is "1".
  Child1 is "B-52␣responded␣to␣call␣for␣airstrike␣from␣SF-Soldiers".
  Parents is "1".
  Parent1 is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".

Visual_object: Node_1111
  Type is node.
  Label is "B-52␣responded␣to␣call␣for␣airstrike␣from␣SF-Soldiers".
  Reference is "(16)".
  Children is "1".
  Child1 is "SF−Soldiers␣called␣in␣second␣airstrike".
  Parents is "1".
  Parent1 is "B−52␣targeted␣SF−Soldiers␣position".

Visual_object: Node_11111
  Type is node.
  Label is "SF−Soldiers␣called␣in␣second␣airstrike".
  Reference is "(15)".
  Children is "2".
  Child1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Child2 is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".

Visual_object: Node_111111
  Type is node.
  Label is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Reference is "(14)".
  Children is "4".
  Child1 is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Child2 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Child3 is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Child4 is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "SF−Soldiers␣called␣in␣second␣airstrike".

Visual_object: Node_1111111
  Type is node.
  Label is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Reference is "(13)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_1111112
  Type is node.
  Label is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and␣
      Taliban␣outpost␣position".
  Reference is "(12)".
  Children is "2".
  Child1 is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Child2 is "GPS␣coordinate␣representation".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_11111121
  Type is node.
  Label is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Reference is "(11)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣
      and␣Taliban␣outpost␣position".

Visual_object: Node_1111113
  Type is node.
  Label is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Reference is "(10)".
  Children is "1".
  Child1 is "SF−Soldiers␣changed␣batteries".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
```

```
Visual_object: Node_11111131
  Type is node.
  Label is "SF–Soldiers␣changed␣batteries".
  Reference is "(9)".
  Children is "1".
  Child1 is "Battery␣of␣GPS␣receiver␣died".
  Parents is "1".
  Parent1 is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".

Visual_object: Node_111111311
  Type is node.
  Label is "Battery␣of␣GPS␣receiver␣died".
  Reference is "(8)".
  Parents is "1".
  Parent1 is "SF–Soldiers␣changed␣batteries".

Visual_object: Node_1111114
  Type is node.
  Label is "F/A–18␣successfully␣targeted␣Taliban␣Outpost".
  Reference is "(7)".
  Children is "1".
  Child1 is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
  Parents is "1".
  Parent1 is "SF–Soldiers␣believed␣GPS–coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_11111141
  Type is node.
  Label is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
  Reference is "(6)".
  Children is "1".
  Child1 is "SF–Soldiers␣called␣for␣first␣airstrike".
  Parents is "1".
  Parent1 is "F/A–18␣successfully␣targeted␣Taliban␣Outpost".

Visual_object: Node_111111411
  Type is node.
  Label is "SF–Soldiers␣called␣for␣first␣airstrike".
  Reference is "(5)".
  Children is "1".
  Child1 is "SF–Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Parents is "1".
  Parent1 is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".

Visual_object: Node_1111114111
  Type is node.
  Label is "SF–Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Reference is "(4)".
  Children is "1".
  Child1 is "SF–Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "SF–Soldiers␣called␣for␣first␣airstrike".

Visual_object: Node_112
  Type is node.
  Label is "B–52␣released␣satellite␣guided␣bomb".
  Reference is "(a)".
  Parents is "1".
  Parent1 is "SF–Soldiers␣hit␣by␣2k–pound␣satellite␣guided␣bomb".

Visual_object: Node_111112
```

```
  Type is node.
  Label is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Reference is "(b)".
  Parents is "2".
  Parent1 is "SF−Soldiers␣called␣in␣second␣airstrike".
  Parent2 is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".

Visual_object: Node_11111122
  Type is node.
  Label is "GPS␣coordinate␣representation".
  Parents is "1".
  Parent1 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣
      and␣Taliban␣outpost␣position".

// a dummy node without children.

Visual_object: Node_wo_children
  Type is node.
  Label is "node_wo_children".

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The edges between the nodes
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Edge_11−1
  Type is edge.
  Source is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Target is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".

Visual_object: Edge_111−11
  Type is edge.
  Source is "B−52␣targeted␣SF−Soldiers␣position".
  Target is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".

Visual_object: Edge_1111−111
  Type is edge.
  Source is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Target is "B−52␣targeted␣SF−Soldiers␣position".

Visual_object: Edge_11111−1111
  Type is edge.
  Source is "SF−Soldiers␣called␣in␣second␣airstrike".
  Target is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".

Visual_object: Edge_111111−11111
  Type is edge.
  Source is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Target is "SF−Soldiers␣called␣in␣second␣airstrike".

Visual_object: Edge_1111111−111111
  Type is edge.
  Source is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Edge_1111112−111111
  Type is edge.
  Source is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
```

```
Visual_object: Edge_11111121−1111112
  Type is edge.
  Source is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Target is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".

Visual_object: Edge_11111122−1111112
  Type is edge.
  Source is "GPS␣coordinate␣representation".
  Target is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".

Visual_object: Edge_1111113−111111
  Type is edge.
  Source is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Edge_11111131−1111113
  Type is edge.
  Source is "SF−Soldiers␣changed␣batteries".
  Target is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".

Visual_object: Edge_111111311−111111311
  Type is edge.
  Source is "Battery␣of␣GPS␣receiver␣died".
  Target is "SF−Soldiers␣changed␣batteries".

Visual_object: Edge_1111114−111111
  Type is edge.
  Source is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Edge_11111141−1111114
  Type is edge.
  Source is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Target is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".

Visual_object: Edge_111111411−11111141
  Type is edge.
  Source is "SF−Soldiers␣called␣for␣first␣airstrike".
  Target is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".

Visual_object: Edge_1111114111−111111411
  Type is edge.
  Source is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Target is "SF−Soldiers␣called␣for␣first␣airstrike".

Visual_object: Edge_111112−1111114111
  Type is edge.
  Source is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Target is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".

Visual_object: Edge_111112−11111
  Type is edge.
  Source is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Target is "SF−Soldiers␣called␣in␣second␣airstrike".

Visual_object: Edge_112−11
  Type is edge.
  Source is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Target is "B-52␣released␣satellite␣guided␣bomb".

Visual_object: Edge_1111114111 -11111141
  Type is edge.
  Source is "SF-Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Target is "F/A-18␣responded␣to␣first␣call␣for␣airstrike".

//
    ************************************************************************

// Top Level Methods
//
    ************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Create WB-Graph:
// this method is used to get the tasks described above done one by one
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Create wb_graph
  Step 1. Store Task1 under <current_task_name>.
  Step 2. Decide:
          If <current_task_name> is None, Then
            Delete <current_task>;
            Delete <current_task_name>;
            Return_with_goal_accomplished.
  Step 3. Get_task_item_whose Name is <current_task_name>
            and_store_under <current_task>.
  Step 4. Accomplish_goal: Perform graph_actions.
  Step 5. Store Next of <current_task> under <current_task_name>.
  Step 6. Goto 2.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Perform Graph Actions:
// the selection rule that determines the method to be invoked due to the
// current task's goal.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Selection_rules_for_goal: Perform graph_actions
  If Type of <current_task> is add_ncf, Then
    Accomplish_goal: Add ncf using
      Parent of <current_task>, and Label of <current_task>,
      and Reference of <current_task>, and Node_number of <current_task>.
  If Type of <current_task> is add_new_node, Then
    Accomplish_goal: Add new_node using
      Parent of <current_task>, and Label of <current_task>,
      and Reference of <current_task>, and Node_number of <current_task>.
  If Type of <current_task> is delete_node, Then
    Accomplish_goal: Remove node using
      Label of <current_task>.
  If Type of <current_task> is insert_ncf, Then
    Accomplish_goal: Insert node using
      Parent of <current_task>, and Label of <current_task>,
      and Reference of <current_task>, and Child1 of <current_task>,
      and Child2 of <current_task>, and Child3 of <current_task>,
      and Child4 of <current_task>.
  If Type of <current_task> is merge_nodes, Then
```

```
    Accomplish_goal: Merge nodes using
      Label1 of <current_task>, and Label2 of <current_task>,
      and New of <current_task>, and Child1 of <current_task>,
      and Child2 of <current_task>, and Child3 of <current_task>,
      and Child4 of <current_task>, and Parent1 of <current_task>,
      and Parent2 of <current_task>, and Parent3 of <current_task>,
      and Parent4 of <current_task>.
  If Type of <current_task> is reindex_node, Then
    Accomplish_goal: Reindex node using
      Label of <current_task>.
  If Type of <current_task> is split_node, Then
    Accomplish_goal: Split node using
      Label of <current_task>, and New_label1 of <current_task>,
      and Reference1 of <current_task>, and New2 of <current_task>,
      and Reference2 of <current_task>, and Child1 of <current_task>,
      and Child2 of <current_task>, and Child3 of <current_task>,
      and Child4 of <current_task>, and Parent1 of <current_task>,
      and Parent2 of <current_task>, and Parent3 of <current_task>,
      and Parent4 of <current_task>.
  If Type of <current_task> is alter_node_description, Then
    Accomplish_goal: Alter node_description using
      Label of <current_task>, and New of <current_task>,
      and Reference of <current_task>.
  If Type of <current_task> is extend_node_description, Then
    Accomplish_goal: Extend node_description using
      Label of <current_task>, and Extension of <current_task>.
  If Type of <current_task> is add_edge, Then
    Accomplish_goal: Add edge using
      Source of <current_task>, and Target of <current_task>.
  If Type of <current_task> is alter_edge, Then
    Accomplish_goal: Alter Edge using
      Change of <current_task>, and Source of <current_task>,
      and Target of <current_task>, and New of <current_task>.
  If Type of <current_task> is delete_edge, Then
    Accomplish_goal: Remove edge using
      Source of <current_task>, and Target of <current_task>.
  If Type of <current_task> is reverse_edge, Then
    Accomplish_goal: Reverse edge using
      Source of <current_task>, and Target of <current_task>.
  If Type of <current_task> is add_fact, Then
    Accomplish_goal: Add fact using
      Description of <current_task>.
  If Type of <current_task> is determine_fact, Then
    Accomplish_goal: Determine fact.
  If Type of <current_task> is reindex_whole_graph, Then
    Accomplish_goal: Reindex graph.
  Return_with_goal_accomplished.

//
    ****************************************************************************

// Graph Operations
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Change Parent:
// change the target of the edges between nodes with the given source labels
// and the target parent to the new parent node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
    *

Method_for_goal: Change parent using
                 <cp_old_parent>, and <cp_new_parent>, and <cp_child1>,
                 and <cp_child2>, and <cp_child3>, and <cp_child4>
  Step 1. Decide:
          If <cp_child1> is nil, Then
            Goto 6;
          If <cp_child2> is nil, Then
            Goto 5;
          If <cp_child3> is nil, Then
            Goto 4;
          If <cp_child4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Alter edge using
                           target, and <cp_child4>, and <cp_old_parent>,
                           and <cp_new_parent>.
  Step 3. Accomplish_goal: Alter edge using
                           target, and <cp_child3>, and <cp_old_parent>,
                           and <cp_new_parent>.
  Step 4. Accomplish_goal: Alter edge using
                           target, and <cp_child2>, and <cp_old_parent>,
                           and <cp_new_parent>.
  Step 5. Accomplish_goal: Alter edge using
                           target, and <cp_child1>, and <cp_old_parent>,
                           and <cp_new_parent>.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Change Child:
// change the source of the edges between nodes with the given target labels
// and the source child node to the new child node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Change child using
                 <cc_old_child>, and <cc_new_child>,
                 and <cc_parent1>, and <cc_parent2>,
                 and <cc_parent3>, and <cc_parent4>
  Step 1. Decide:
          If <cc_parent1> is nil, Then
            Goto 6;
          If <cc_parent2> is nil, Then
            Goto 5;
          If <cc_parent3> is nil, Then
            Goto 4;
          If <cc_parent4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Alter edge using
                           source, and <cc_old_child>, and <cc_parent4>,
                           and <cc_new_child>.
  Step 3. Accomplish_goal: Alter edge using
                           source, and <cc_old_child>, and <cc_parent3>,
                           and <cc_new_child>.
  Step 4. Accomplish_goal: Alter edge using
                           source, and <cc_old_child>, and <cc_parent2>,
                           and <cc_new_child>.
  Step 5. Accomplish_goal: Alter edge using
                           source, and <cc_old_child>, and <cc_parent1>,
                           and <cc_new_child>.
  Step 6. Return_with_goal_accomplished.
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add as Parent:
// add edges directing from the given child nodes to the parent node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add_as parent using
                 <aap_parent>, and <aap_child1>,
                 and <aap_child2>, and <aap_child3>,
                 and <aap_child4>
  Step 1. Decide:
          If <aap_child1> is nil, Then
            Goto 6;
          If <aap_child2> is nil, Then
            Goto 5;
          If <aap_child3> is nil, Then
            Goto 4;
          If <aap_child4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Add edge using
                           <aap_child4>, and <aap_parent>.
  Step 3. Accomplish_goal: Add edge using
                           <aap_child3>, and <aap_parent>.
  Step 4. Accomplish_goal: Add edge using
                           <aap_child2>, and <aap_parent>.
  Step 5. Accomplish_goal: Add edge using
                           <aap_child1>, and <aap_parent>.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add as Child:
// add edges directing form the child node to the given parent nodes
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add_as child using
                 <aac_child>, and <aac_parent1>,
                 and <aac_parent2>, and <aac_parent3>,
                 and <aac_parent4>
  Step 1. Decide:
          If <aac_parent1> is nil, Then
            Goto 6;
          If <aac_parent2> is nil, Then
            Goto 5;
          If <aac_parent3> is nil, Then
            Goto 4;
          If <aac_parent4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent4>.
  Step 3. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent3>.
  Step 4. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent2>.
  Step 5. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent1>.
  Step 6. Return_with_goal_accomplished.
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Reindex Graph:
// issue the reindex whole graph command
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Reindex graph
  Step 1. Accomplish_goal: Issue context_command using
                           "reindex␣whole␣graph", and canvas,
                           and nil, and nil.
  Step 2. Return_with_goal_accomplished.

//
    ************************************************************************

// Node Operations
//
    ************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Necessary Causal Factor (NCF):
// issue the command to add a ncf
// enter the node description given in actual task
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add ncf using
                 <ncf_parent>, and <ncf_label>,
                 and <ncf_reference>, and <ncf_node_number>
  Step 1. Accomplish_goal: Issue context_command using
                           "add␣ncf", and node,
                           and <ncf_parent>, and nil.
  Step 2. Accomplish_goal: Enter ncf_description using
                           <ncf_label>, and <ncf_reference>,
                           and <ncf_node_number>.
  Step 3. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Node:
// issue the command to add a new node
// enter the node description given in actual task
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add new_node using
                 <nn_parent>, and <nn_label>,
                 and <nn_reference>, and <nn_node_number>
  Step 1. Accomplish_goal: Issue context_command using
                           "add␣new␣node", and canvas,
                           and nil, and nil.
  Step 2. Accomplish_goal: Enter new_node_description using
                           <nn_label>, and <nn_reference>,
                           and <nn_node_number>.
  Step 3. Return_with_goal_accomplished.
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Remove Node:
// find the node to delete
// if the node is not a leaf node:
//   - delete all the nodes edges to all child nodes
// issue the delete command
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Remove node using <rm_label>
  Step 1. Look_for_object_whose Label is <rm_label> and_store_under <target>.
  Step 2. Decide:
          If Children of <target> is nil, Then
            Goto 7;
          If Children of <target> is "1", Then
            Goto 6;
          If Children of <target> is "2", Then
            Goto 5;
          If Children of <target> is "3", Then
            Goto 4.
  Step 3. Accomplish_goal: Remove edge using
                           Child4 of <target>, and <rm_label>.
  Step 4. Accomplish_goal: Remove edge using
                           Child3 of <target>, and <rm_label>.
  Step 5. Accomplish_goal: Remove edge using
                           Child2 of <target>, and <rm_label>.
  Step 6. Accomplish_goal: Remove edge using
                           Child1 of <target>, and <rm_label>.
  Step 7. Accomplish_goal: Issue context_command using
                           "delete_node", and <rm_label>, and nil, and nil.
  Step 8. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Insert Node: (between parent node and optional child nodes)
// issue the command to add a necessary causal factor of the parent node
// alter edges between parent node and given child nodes: new parent is new
    node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Insert node using
                 <in_parent>, and <in_label>,
                 and <in_reference>, and <in_child1>,
                 and <in_child2>, and <in_child3>,
                 and <in_child4>
  Step 1. Accomplish_goal: Add ncf using
                           <in_parent>, and <in_label>,
                           and <in_reference>, and nil.
  Step 2. Accomplish_goal: Change parent using
                           <in_parent>, and <in_label>,
                           and <in_child1>, and <in_child2>,
                           and <in_child3>, and <in_child4>.
  Step 3. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Merge Nodes: (take care of corresponding visual objects!)
// change label of first node to the new label
// alter edges between second node and given parent nodes: new child is first
// node
// alter edges between second node and given child nodes: new parent is first
// node
// delete the second node (now without children: dummy visual object)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Merge nodes using
                 <mn_label1>, and <mn_label2>,
                 and <mn_new_label>, and <mn_new_rewference>,
                 and <mn_child1>, and <mn_child2>,
                 and <mn_child3>, and <mn_child4>,
                 and <mn_parent1>, and <mn_parent2>,
                 and <mn_parent3>, and <mn_parent4>
  Step 1. Accomplish_goal: Alter node_description using
                           <mn_label1>, and <mn_new_label>,
                           and <mn_new_reference>.
  Step 2. Accomplish_goal: Change child using
                           <mn_label2>, and <mn_new_label>,
                           and <mn_parent1>, and <mn_parent2>,
                           and <mn_parent3>, and <mn_parent4>.
  Step 3. Accomplish_goal: Change parent using
                           <mn_label2>, and <mn_new_label>,
                           and <mn_child1>, and <mn_child2>,
                           and <mn_child3>, and <mn_child4>.
  Step 4. Accomplish_goal: Remove node using
                           "node_wo_children".
  Step 5. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Reindex Node:
// issue the reindex node command
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Reindex node using
                 <ri_label>
  Step 1. Accomplish_goal: Issue context_command using
                           "reindex␣node", and <ri_label>,
                           and nil, and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Split Node: (take care of corresponding visual objects!)
// change label of the node to the one of the new labels and reference
// add a NCF of the parent of the original node with the other of the new
    labels
// and reference
// add edges between given parent nodes: new child is new node
// add edges between given child nodes: new parent is new node
// node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Split node using
                 <sn_label>, and <sn_new_label1>,
                 and <sn_reference1>, and <sn_new_label2>,
                 and <sn_reference2>, and <sn_child1>,
                  and <sn_child2>, and <sn_child3>,
                  and <sn_child4>, and <sn_parent1>,
                  and <sn_parent2>, and <sn_parent3>,
                  and <sn_parent4>
  Step 1. Accomplish_goal: Alter node_description using
                           <sn_label>, and <sn_new_label1>,
                           and <sn_reference1>.
  Step 2. Accomplish_goal: Add ncf using
                           <sn_parent1>, and <sn_new_label2>,
                           and <sn_reference2>, and nil.
  Step 3. Accomplish_goal: Add_as child using
                           <sn_new_label2>, and <sn_parent1>,
                           and <sn_parent2>, and <sn_parent3>,
                           and <sn_parent4>.
  Step 4. Accomplish_goal: Add_as parent using
                           <sn_new_label2>, and <sn_child1>,
                           and <sn_child2>, and <sn_child3>,
                           and <sn_child4>.
  Step 5. Return_with_goal_accomplished.

//
    ****************************************************************************

// Node-Description Operators
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Alter Node Description:
// issue the edit node command
// select the text input field and the text
// enter the new node description
// close the dialog box (hit 'return'-key)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Alter node_description using
                 <and_original_label>, and <and_new_label>,
                 <and_reference>
  Step 1. Accomplish_goal: Issue context_command using
                           "edit_node", and node,
                           and <and_original_label>, and nil.
  Step 2. Accomplish_goal: Double_click_at item using
                           "Label", and input_field.
  Step 3. Accomplish_goal: Enter node_text using
                           <and_new_label>.
  Step 4. Accomplish_goal: Enter node_text using
                           <and_reference>.
  Step 5. Keystroke CR.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter NCF Description:
// if the node number is given:
//   -select the number input field
//   -enter the node number
//   -select the text input field
// enter the node text
// enter the node reference to List of Facts item
// close the dialog box (dummy: type in "X"  representing a 'return'-key hit)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter ncf_description using
                 <encfd_label>, and <encfd_reference>,
                 and <encfd_number>
  Step 1. Decide:
          If <encfd_number> is nil, Then
            Goto 5.
  Step 2. Accomplish_goal: Double_click_at item using
                           "Path", and input_field.
  Step 3. Accomplish_goal: Enter node_number using
                           <encfd_number>.
  Step 4. Accomplish_goal: Double_click_at item using
                           "Label", and input_field.
  Step 5. Accomplish_goal: Enter node_text using
                           <encfd_label>.
  Step 6. Accomplish_goal: Enter node_text using
                           <encfd_reference>.
  Step 7. Keystroke CR.
  Step 8. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter New Node Description:
// if the node number is given: enter the node number
// select the text input field
// enter the node text
// enter the node reference to List of Facts item
// close the dialog box (hit 'return'-key)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter new_node_description using
                 <ennd_label>, and <ennd_reference>,
                 and <ennd_number>
  Step 1. Decide:
          If <ennd_number> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Enter node_number using
                           <ennd_number>.
  Step 3. Accomplish_goal: Double_click_at item using
                           "Label", and input_field.
  Step 4. Accomplish_goal: Enter node_text using
                           <ennd_label>.
  Step 5. Accomplish_goal: Enter node_text using
                           <ennd_reference>.
  Step 6. Keystroke CR.
  Step 7. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter Node Number:
// type in the node number
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter node_number using
                 <enn_number>
  Step 1. Type_in <enn_number>.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter Node Text:
// type in the node description text
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter node_text using
                 <ent_text>
  Step 1. Decide:
          If <ent_text> is_not nil, Then
            Type_in <ent_text>.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Extend Node Description:
// issue the edit node command
// select the text input field
// enter the new node description
// close the dialog box (hit 'return'-key)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Extend node_description using
                 <xtnd_original_label>, and <xtnd_new_label>
  Step 1. Accomplish_goal: Issue context_command using
                           "edit␣node", and node,
                           and <xtnd_original_label>, and nil.
  Step 2. Accomplish_goal: Click_at item using
                           "Label", and input_field.
  Step 3. Accomplish_goal: Enter node_text using
                           <xtnd_new_label>.
  Step 4. Keystroke CR.
  Step 5. Return_with_goal_accomplished.

//
    ***************************************************************************
// Edge Operations
//
    ***************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Edge:
// drag from the source node to the targetnode
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add edge using
                 <ae_source>, and <ae_target>
  Step 1. Accomplish_goal: Drag item using
                           <ae_source>, and node,
                           and <ae_target>, and node.
  Step 2. Return_with_goal_accomplished.
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Alter Edge:
// determine, whether the source or the target of the edge is to be changed
// either change the source of the edge
// or
// add an edge from source node to the new target node of the edge
// and delete the old edge
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Alter edge using
                 <ale_change>, and <ale_source>, and <ale_target>, and <ale_new
                     >
  Step 1. Decide:
          If <ale_change> is target, Then
            Goto 4.
  Step 2. Accomplish_goal: Drag edge using
                           <ale_source>, and <ale_target>,
                           and <ale_new>.
  Step 3. Return_with_goal_accomplished.
  Step 4. Accomplish_goal: Add edge using
                           <ale_source>, and <ale_new>.
  Step 5. Accomplish_goal: Remove edge using
                           <ale_source>, and <ale_target>.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Remove Edge:
// issue the delete command in the context sensitive menu
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Remove edge using
                 <rme_source>, and <rme_target>
  Step 1. Accomplish_goal: Issue context_command using
                           "delete␣edge", and edge,
                           and <rme_source>, and <rme_target>.
  Step 2. Return_with_goal_accomplished.

//
    ****************************************************************************
// List of Facts Operations
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Fact:
// switch to the text editor
// find and click on the insertion point (dummy)
// enter the index number
// enter the fact description
// switch to yb-edit
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
```

```
Method_for_goal: Add fact using
                <af_description>
  Step 1. Accomplish_goal: Change_to program using
                          "Editor".
  Step 2. Accomplish_goal: Click_at item using
                          nil, and "insertion_point".
  Step 3. Accomplish_goal: Enter fact_description using
                          <af_description>.
  Step 4. Accomplish_goal: Enter index_number.
  Step 5. Accomplish_goal: Change_to program using
                          "YB-Edit".
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Determine Fact:
// switch to the text editor
// find next fact (dummy: a verify operator which consumes search/decision time
    )
// change to yb-edit
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Determine fact
  Step 1. Accomplish_goal: Change_to program using
                          "Editor".
  Step 2. Verify "Find␣the␣fact␣that␣is␣causal␣factor".
  Step 3. Accomplish_goal: Change_to program using
                          "YB-Edit".
  Step 4. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter Fact Description:
// type in description text
// check description text
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter fact_description using
                <ef_description>
  Step 1. Type_in <ef_description>.
  Step 2. Verify "Description␣is␣correct".
  Step 3. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Enter Index Number:
// think about index number (dummy: X)
// type in index number (dummy: X)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Enter index_number
  Step 1. Think_of "X".
  Step 2. Type_in "(X)".
  Step 3. Return_with_goal_accomplished.

//
```

```
    ************************************************************************
// Program Operations
//
    ************************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Change_to Program:
// klick program window
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Change_to program using
                <ctp_program_name>
  Step 1. Accomplish_goal: Click_at item using
                          <ctp_program_name>, and window.
  Step 2. Return_with_goal_accomplished.

//
    ************************************************************************
// Basic Operations
//
    ************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Click at Edge:
// find the egde
// point to the edge
// click mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Click_at edge using
                <cae_source>, and <cae_target>
  Step 1. Look_for_object_whose Type is edge, and Source is <cae_source>,
          and Target is <cae_target> and_store_under <cae_edge>.
  Step 2. Point_to <cae_edge>.
  Step 3. Click mouse_button.
  Step 4. Delete <cae_edge>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Click at Item:
// find an item with a specific label and a specific type
// point to the item
// click mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Click_at item using
                <cai_label>, and <cai_type>
  Step 1. Look_for_object_whose Label is <cai_label>,
          and Type is <cai_type> and_store_under <cai_target>.
  Step 2. Point_to <cai_target>.
  Step 3. Click mouse_button.
  Step 4. Delete <cai_target>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Drag Edge:
// find the edge
// point to the edge
// hold down the mouse button
// find the target
// point to the target
// release mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Drag edge using
                 <da_edge_source>, and <da_edge_target>,
                 and <da_destination>
  Step 1. Look_for_object_whose Type is edge, and Source is <da_edge_source>,
            and Target is <da_edge_target> and_store_under <da_edge>.
  Step 2. Point_to <da_edge>.
  Step 3. Hold_down mouse_button.
  Step 4. Look_for_object_whose Type is node, and Label is <da_destination>
            and_store_under <da_target>.
  Step 5. Point_to <da_target>.
  Step 6. Release mouse_button.
  Step 7. Delete <da_edge>; Delete <da_target>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Drag Item:
// find the object
// point to the object
// hold down the mouse button
// find the target
// point to the target
// release mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Drag item using
                 <di_label>, and <di_type>,
                 and <di_destination>, and <di_dest_type>
  Step 1. Look_for_object_whose Type is <di_type>, and Label is <di_label>
            and_store_under <di_item>.
  Step 2. Point_to <di_item>.
  Step 3. Hold_down mouse_button.
  Step 4. Look_for_object_whose Type is <di_dest_type>,
            and Label is <di_destination> and_store_under <di_target>.
  Step 5. Point_to <di_target>.
  Step 6. Release mouse_button.
  Step 7. Delete <di_item>; Delete <di_target>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Double Click at Item:
// find an item with a specific label and a specific type
// point to the item
// double click mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
```

```
Method_for_goal: Double_click_at item using
                 <dci_label>, and <dci_type>
  Step 1. Look_for_object_whose Label is <dci_label>,
            and Type is <dci_type> and_store_under <dci_target>.
  Step 2. Point_to <dci_target>.
  Step 3. Double_click mouse_button.
  Step 4. Delete <dci_target>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *
// Issue Context Command:
// Locate and
// (right-)click on the corresponding object
// Find and click the corresponding menu entry
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *

Method_for_goal: Issue context_command using
                 <icc_command>, and <icc_context>,
                 and <icc_label1>, and <icc_label2>
  Step 1. Decide:
          If <icc_context> is canvas, Then
            Accomplish_goal: Click_at item using
                             nil, and <icc_context>;
          If <icc_context> is edge, Then
            Accomplish_goal: Click_at edge using
                             <icc_label1>, and <icc_label2>;
          If <icc_context> is node, Then
            Accomplish_goal: Click_at item using
                             <icc_label1>, and <icc_context>.
  Step 2. Accomplish_goal: Click_at item using
                           <icc_command>, and menu_entry.
  Step 3. Return_with_goal_accomplished.
```

C. GOMSL Model of the New Design

```
Define_model: "new␣yb−edit␣use␣GPS␣Friendly␣Fire"
  Starting_goal is Create wb_graph.

//
    ******************************************************************************
// Assumptions:
//   − a List of Facts exists
//   − the new yb−edit is allready running, presenting a new empty document
//      and displaying the List of Facts
//   − the canvas's size is big enough to display the whole graph :−)
//   − nodes have no more than four child and four parent nodes (this must also
//      true for the result nodes of merge operations!)
//
    ******************************************************************************

//
    ******************************************************************************
// The Tasks
// a list of task items
//
    ******************************************************************************

Task_item: Task1
  Name is Task1.
  Type is determine_fact.
  Next is Task2.

Task_item: Task2
  Name is Task2.
  Type is add_new_node.
  Label is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
  Node_number is "1".
  Reference is "(1+2)".
  Next is Task3.

Task_item: Task3
  Name is Task3.
  Type is determine_fact.
  Next is Task4.

Task_item: Task4
  Name is Task4.
  Type is add_ncf.
  Label is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Reference is "(3)".
```

```
    Parent is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
    Next is Task5.

  Task_item: Task5
    Name is Task5.
    Type is determine_fact.
    Next is Task6.

  Task_item: Task6
    Name is Task6.
    Type is add_ncf.
    Label is "B−52␣targeted␣SF−Soldiers␣position".
    Reference is "(17)".
    Parent is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
    Next is Task7.

  Task_item: Task7
    Name is Task7.
    Type is determine_fact.
    Next is Task8.

  Task_item: Task8
    Name is Task8.
    Type is add_ncf.
    Label is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
    Reference is "(16)".
    Parent is "B−52␣targeted␣SF−Soldiers␣position".
    Next is Task9.

  Task_item: Task9
    Name is Task9.
    Type is determine_fact.
    Next is Task10.

  Task_item: Task10
    Name is Task10.
    Type is add_ncf.
    Label is "SF−Soldiers␣called␣in␣second␣airstrike".
    Reference is "(15)".
    Parent is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
    Next is Task11.

  Task_item: Task11
    Name is Task11.
    Type is determine_fact.
    Next is Task12.

  Task_item: Task12
    Name is Task12.
    Type is add_ncf.
    Label is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
    Reference is "(14)".
    Parent is "SF−Soldiers␣called␣in␣second␣airstrike".
    Next is Task13.

  Task_item: Task13
    Name is Task13.
    Type is determine_fact.
    Next is Task14.

  Task_item: Task14
    Name is Task14.
    Type is add_ncf.
    Label is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
    Reference is "(13)".
    Parent is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
    Next is Task15.

Task_item: Task15
   Name is Task15.
   Type is determine_fact.
   Next is Task16.

Task_item: Task16
   Name is Task16.
   Type is add_ncf.
   Label is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and␣
       Taliban␣outpost␣position".
   Reference is "(12)".
   Parent is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
   Next is Task17.

Task_item: Task17
   Name is Task17.
   Type is determine_fact.
   Next is Task18.

Task_item: Task18
   Name is Task18.
   Type is add_ncf.
   Label is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
       seconds␣of␣lat/long".
   Reference is "(11)".
   Parent is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
       ␣Taliban␣outpost␣position".
   Next is Task19.

Task_item: Task19
   Name is Task19.
   Type is determine_fact.
   Next is Task20.

Task_item: Task20
   Name is Task20.
   Type is add_ncf.
   Label is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
   Reference is "(10)".
   Parent is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
   Next is Task21.

Task_item: Task21
   Name is Task21.
   Type is determine_fact.
   Next is Task22.

Task_item: Task22
   Name is Task22.
   Type is add_ncf.
   Label is "SF−Soldiers␣changed␣batteries".
   Reference is "(9)".
   Parent is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
   Next is Task23.

Task_item: Task23
   Name is Task23.
   Type is determine_fact.
```

```
    Next is Task24.

  Task_item: Task24
    Name is Task24.
    Type is add_ncf.
    Label is "Battery␣of␣GPS␣receiver␣died".
    Reference is "(8)".
    Parent is "SF–Soldiers␣changed␣batteries".
    Next is Task25.

  Task_item: Task25
    Name is Task25.
    Type is determine_fact.
    Next is Task26.

  Task_item: Task26
    Name is Task26.
    Type is add_ncf.
    Label is "F/A–18␣successfully␣targeted␣Taliban␣Outpost".
    Reference is "(7)".
    Parent is "SF–Soldiers␣believed␣GPS–coords␣to␣be␣Taliban␣Outpost".
    Next is Task27.

  Task_item: Task27
    Name is Task27.
    Type is determine_fact.
    Next is Task28.

  Task_item: Task28
    Name is Task28.
    Type is add_ncf.
    Label is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
    Reference is "(6)".
    Parent is "F/A–18␣successfully␣targeted␣Taliban␣Outpost".
    Next is Task29.

  Task_item: Task29
    Name is Task29.
    Type is determine_fact.
    Next is Task30.

  Task_item: Task30
    Name is Task30.
    Type is add_ncf.
    Label is "SF–Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".
    Reference is "(4)".
    Parent is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
    Next is Task31.

  Task_item: Task31
    Name is Task31.
    Type is determine_fact.
    Next is Task32.

  Task_item: Task32
    Name is Task32.
    Type is insert_ncf.
    Label is "SF–Soldiers␣called␣for␣first␣airstrike".
    Reference is "(5)".
    Parent is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
    Child1 is "SF–Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".
    Next is Task33.

  // extending node description with the fact reference is not required anymore,
  // because the new yb-edit manages the List of Facts. Now, the fact is added
  // to the list of facts first, and afterwards this fact added as a ncf to a
  // node.

  Task_item: Task33
    Name is Task33.
    Type is add_fact.
    Description is "B-52␣released␣satellite␣guided␣bomb".
    Next is Task34.

  Task_item: Task34
    Name is Task34.
    Type is add_ncf.
    Label is "B-52␣released␣satellite␣guided␣bomb".
    Parent is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".
    Next is Task35.

  Task_item: Task35
    Name is Task35.
    Type is add_fact.
    Description is "SF-Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
    Next is Task36.

  Task_item: Task36
    Name is Task36.
    Type is add_ncf.
    Label is "SF-Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
    Parent is "SF-Soldiers␣called␣in␣second␣airstrike".
    Next is Task37.

  Task_item: Task37
    Name is Task37.
    Type is add_edge.
    Source is "SF-Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
    Target is "SF-Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".
    Next is Task38.

  Task_item: Task38
    Name is Task38.
    Type is add_fact.
    Description is "GPS␣coordinate␣representation".
    Next is Task39.

  Task_item: Task39
    Name is Task39.
    Type is add_ncf.
    Label is "GPS␣coordinate␣representation".
    Parent is "SF-Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
        ␣Taliban␣outpost␣position".
    Next is None.

  // adding the reference to the node description was forgotten
  // not any longer: automaticly handled by yb-edit!

  //
      ************************************************************************

  // Visual Objects
  //
```

```
    ***************************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *
// The Program Windows
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *

Visual_object: Ybedit_program_window
  Type is window.
  Label is "YB-Edit".

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *
// new YB-Edits interface elements
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   *

Visual_object: Canvas
  Type is canvas.
  Label is "Graph".

Visual_object: List_of_facts
  Type is list.
  Label is "List␣of␣Facts".

Visual_object: Fact_description
  Type is input_field.
  Label is "Fact␣Description".

Visual_object: Fact_reference
  Type is input_field.
  Label is "Fact␣Reference".

Visual_object: Node_Description
  Type is input_field.
  Label is "Node␣Description".

Visual_object: Node_Reference
  Type is input_field.
  Label is "Node␣Reference".

Visual_object: Causes
  Type is input_field.
  Label is "Causes".

Visual_object: Effects
  Type is input_field.
  Label is "Effects".

Visual_object: Trash
  Type is trash.
  Label is "Trash".

Visual_object: Merge
  Type is field.
  Label is "Merge".

Visual_object: Split
  Type is field.
  Label is "Split".
```

```
// a dummy selection

Visual_object: Selection
  Type is selection.

// a dummy insertion point

Visual_object: Insertion_point
  Type is insertion_point.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The nodes representing the graph
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Node_1
  Type is node.
  Label is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".
  Reference is "(1+2)".
  Children is "1".
  Child1 is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".

Visual_object: Node_11
  Type is node.
  Label is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Reference is "(3)".
  Children is "2".
  Child1 is "B−52␣targeted␣SF−Soldiers␣position".
  Child2 is "B−52␣released␣satellite␣guided␣bomb".
  Parents is "1".
  Parent1 is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".

Visual_object: Node_111
  Type is node.
  Label is "B−52␣targeted␣SF−Soldiers␣position".
  Reference is "(17)".
  Children is "1".
  Child1 is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Parents is "1".
  Parent1 is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".

Visual_object: Node_1111
  Type is node.
  Label is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Reference is "(16)".
  Children is "1".
  Child1 is "SF−Soldiers␣called␣in␣second␣airstrike".
  Parents is "1".
  Parent1 is "B−52␣targeted␣SF−Soldiers␣position".

Visual_object: Node_11111
  Type is node.
  Label is "SF−Soldiers␣called␣in␣second␣airstrike".
  Reference is "(15)".
  Children is "2".
  Child1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Child2 is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".

Visual_object: Node_111111
  Type is node.
  Label is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Reference is "(14)".
  Children is "4".
  Child1 is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Child2 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Child3 is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Child4 is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "SF−Soldiers␣called␣in␣second␣airstrike".

Visual_object: Node_1111111
  Type is node.
  Label is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Reference is "(13)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_1111112
  Type is node.
  Label is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and␣
      Taliban␣outpost␣position".
  Reference is "(12)".
  Children is "2".
  Child1 is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Child2 is "GPS␣coordinate␣representation".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_11111121
  Type is node.
  Label is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Reference is "(11)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣
      and␣Taliban␣outpost␣position".

Visual_object: Node_1111113
  Type is node.
  Label is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Reference is "(10)".
  Children is "1".
  Child1 is "SF−Soldiers␣changed␣batteries".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_11111131
  Type is node.
  Label is "SF−Soldiers␣changed␣batteries".
  Reference is "(9)".
  Children is "1".
  Child1 is "Battery␣of␣GPS␣receiver␣died".
  Parents is "1".
  Parent1 is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".

Visual_object: Node_111111311
  Type is node.
  Label is "Battery␣of␣GPS␣receiver␣died".
  Reference is "(8)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣changed␣batteries".
```

```
Visual_object: Node_1111114
  Type is node.
  Label is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".
  Reference is "(7)".
  Children is "1".
  Child1 is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Parents is "1".
  Parent1 is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Node_11111141
  Type is node.
  Label is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".
  Reference is "(6)".
  Children is "1".
  Child1 is "SF−Soldiers␣called␣for␣first␣airstrike".
  Parents is "1".
  Parent1 is "F/A−18␣successfully␣targeted␣Taliban␣Outpost".

Visual_object: Node_111111411
  Type is node.
  Label is "SF−Soldiers␣called␣for␣first␣airstrike".
  Reference is "(5)".
  Children is "1".
  Child1 is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Parents is "1".
  Parent1 is "F/A−18␣responded␣to␣first␣call␣for␣airstrike".

Visual_object: Node_1111114111
  Type is node.
  Label is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Reference is "(4)".
  Children is "1".
  Child1 is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Parents is "1".
  Parent1 is "SF−Soldiers␣called␣for␣first␣airstrike".

Visual_object: Node_112
  Type is node.
  Label is "B−52␣released␣satellite␣guided␣bomb".
  Reference is "(a)".
  Parents is "1".
  Parent1 is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".

Visual_object: Node_111112
  Type is node.
  Label is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Reference is "(b)".
  Parents is "2".
  Parent1 is "SF−Soldiers␣called␣in␣second␣airstrike".
  Parent2 is "SF−Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".

Visual_object: Node_11111122
  Type is node.
  Label is "GPS␣coordinate␣representation".
  Parents is "1".
  Parent1 is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣
      and␣Taliban␣outpost␣position".

// a dummy node without children.
```

```
Visual_object: Node_wo_children
  Type is node.
  Label is "node_wo_children".

// a dummy new node

Visual_object: New_node
  Type is node.
  Label is "New␣Node".

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// The edges between the nodes
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Edge_11−1
  Type is edge.
  Source is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".
  Target is "Accident:␣3␣SF−Soldiers␣died␣and␣20␣wounded".

Visual_object: Edge_111−11
  Type is edge.
  Source is "B−52␣targeted␣SF−Soldiers␣position".
  Target is "SF−Soldiers␣hit␣by␣2k−pound␣satellite␣guided␣bomb".

Visual_object: Edge_1111−111
  Type is edge.
  Source is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Target is "B−52␣targeted␣SF−Soldiers␣position".

Visual_object: Edge_11111−1111
  Type is edge.
  Source is "SF−Soldiers␣called␣in␣second␣airstrike".
  Target is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".

Visual_object: Edge_111111−11111
  Type is edge.
  Source is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Target is "SF−Soldiers␣called␣in␣second␣airstrike".

Visual_object: Edge_1111111−111111
  Type is edge.
  Source is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Edge_1111112−111111
  Type is edge.
  Source is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".
  Target is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".

Visual_object: Edge_11111121−1111112
  Type is edge.
  Source is "Taliban␣Outpost␣position␣and␣SF−Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Target is "SF−Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
      ␣Taliban␣outpost␣position".

Visual_object: Edge_11111122−1111112
  Type is edge.
  Source is "GPS␣coordinate␣representation".
```

```
    Target is "SF-Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and
        ␣Taliban␣outpost␣position".

  Visual_object: Edge_1111113-111111
    Type is edge.
    Source is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
    Target is "SF-Soldiers␣believed␣GPS-coords␣to␣be␣Taliban␣Outpost".

  Visual_object: Edge_11111131-1111113
    Type is edge.
    Source is "SF-Soldiers␣changed␣batteries".
    Target is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".

  Visual_object: Edge_111111311-111111311
    Type is edge.
    Source is "Battery␣of␣GPS␣receiver␣died".
    Target is "SF-Soldiers␣changed␣batteries".

  Visual_object: Edge_1111114-111111
    Type is edge.
    Source is "F/A-18␣successfully␣targeted␣Taliban␣Outpost".
    Target is "SF-Soldiers␣believed␣GPS-coords␣to␣be␣Taliban␣Outpost".

  Visual_object: Edge_11111141-1111114
    Type is edge.
    Source is "F/A-18␣responded␣to␣first␣call␣for␣airstrike".
    Target is "F/A-18␣successfully␣targeted␣Taliban␣Outpost".

  Visual_object: Edge_111111411-11111141
    Type is edge.
    Source is "SF-Soldiers␣called␣for␣first␣airstrike".
    Target is "F/A-18␣responded␣to␣first␣call␣for␣airstrike".

  Visual_object: Edge_1111114111-111111411
    Type is edge.
    Source is "SF-Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".
    Target is "SF-Soldiers␣called␣for␣first␣airstrike".

  Visual_object: Edge_111112-1111114111
    Type is edge.
    Source is "SF-Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
    Target is "SF-Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".

  Visual_object: Edge_111112-11111
    Type is edge.
    Source is "SF-Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
    Target is "SF-Soldiers␣called␣in␣second␣airstrike".

  Visual_object: Edge_112-11
    Type is edge.
    Source is "SF-Soldiers␣hit␣by␣2k-pound␣satellite␣guided␣bomb".
    Target is "B-52␣released␣satellite␣guided␣bomb".

  Visual_object: Edge_1111114111-11111141
    Type is edge.
    Source is "SF-Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
        device".
    Target is "F/A-18␣responded␣to␣first␣call␣for␣airstrike".

  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      *
```

```
// The facts
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Visual_object: Fact1_2
  Type is fact.
  Label is "Accident:␣3␣SF–Soldiers␣died␣and␣20␣wounded".
  Number is "1+2".

Visual_object: Fact3
  Type is fact.
  Label is "SF–Soldiers␣hit␣by␣2k–pound␣satellite␣guided␣bomb".
  Number is "3".

Visual_object: Fact4
  Type is fact.
  Label is "SF–Soldiers␣aquired␣Taliban␣Outpost␣position␣with␣GPS␣targetting␣
      device".
  Number is "4".

Visual_object: Fact5
  Type is fact.
  Label is "SF–Soldiers␣called␣for␣first␣airstrike".
  Number is "5".

Visual_object: Fact6
  Type is fact.
  Label is "F/A–18␣responded␣to␣first␣call␣for␣airstrike".
  Number is "6".

Visual_object: Fact7
  Type is fact.
  Label is "F/A–18␣successfully␣targeted␣Taliban␣Outpost".
  Number is "7".

Visual_object: Fact8
  Type is fact.
  Label is "Battery␣of␣GPS␣receiver␣died".
  Number is "8".

Visual_object: Fact9
  Type is fact.
  Label is "SF–Soldiers␣changed␣batteries".
  Number is "9".

Visual_object: Fact10
  Type is fact.
  Label is "GPS␣reinits␣with␣own␣position␣after␣power␣outage".
  Number is "10".

Visual_object: Fact11
  Type is fact.
  Label is "Taliban␣Outpost␣position␣and␣SF–Soldiers␣Position␣differ␣only␣in␣
      seconds␣of␣lat/long".
  Number is "11".

Visual_object: Fact12
  Type is fact.
  Label is "SF–Soldiers␣could␣not␣tell␣the␣difference␣between␣own␣position␣and␣
      Taliban␣outpost␣position".
  Number is "12".

Visual_object: Fact13
```

```
  Type is fact.
  Label is "SF−Soldiers␣were␣not␣aware␣on␣GPS␣reinitialisation␣procedures".
  Number is "13".

Visual_object: Fact14
  Type is fact.
  Label is "SF−Soldiers␣believed␣GPS−coords␣to␣be␣Taliban␣Outpost".
  Number is "14".

Visual_object: Fact15
  Type is fact.
  Label is "SF−Soldiers␣called␣in␣second␣airstrike".
  Number is "15".

Visual_object: Fact16
  Type is fact.
  Label is "B−52␣responded␣to␣call␣for␣airstrike␣from␣SF−Soldiers".
  Number is "16".

Visual_object: Fact17
  Type is fact.
  Label is "B−52␣targeted␣SF−Soldiers␣position".
  Number is "17".

Visual_object: Fact18
  Type is fact.
  Label is "B−52␣released␣satellite␣guided␣bomb".
  Number is "18".

Visual_object: Fact19
  Type is fact.
  Label is "SF−Soldiers␣wanted␣to␣attack␣Taliban␣Outpost".
  Number is "19".

Visual_object: Fact20
  Type is fact.
  Label is "GPS␣coordinate␣representation".
  Number is "20".

//
    ****************************************************************************
// Top Level Methods
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Create WB−Graph:
// this method is used to get the tasks described above done one by one
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Create wb_graph
  Step 1. Store Task1 under <current_task_name>.
  Step 2. Decide:
          If <current_task_name> is None, Then
            Delete <current_task>;
            Delete <current_task_name>;
            Return_with_goal_accomplished.
  Step 3. Get_task_item_whose Name is <current_task_name>
            and_store_under <current_task>.
    Step 4. Accomplish_goal: Perform graph_actions.
    Step 5. Store Next of <current_task> under <current_task_name>.
    Step 6. Goto 2.

  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     *
  // Perform Graph Actions:
  // the selection rule that determines the method to be invoked due to the
  // current task's goal.
  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     *

  Selection_rules_for_goal: Perform graph_actions
    If Type of <current_task> is add_new_node, Then
      Accomplish_goal: Add node using
        Label of <current_task>, and canvas, and "Graph".
    If Type of <current_task> is add_ncf, Then
      Accomplish_goal: Add node using
        Label of <current_task>, and node, and Parent of <current_task>.
    If Type of <current_task> is delete_node, Then
      Accomplish_goal: Remove node using
        Label of <current_task>.
    If Type of <current_task> is insert_ncf, Then
      Accomplish_goal: Insert node using
        Parent of <current_task>, and Label of <current_task>,
        and Child1 of <current_task>, and Child2 of <current_task>,
        and Child3 of <current_task>, and Child4 of <current_task>.
    If Type of <current_task> is merge_nodes, Then
      Accomplish_goal: Merge nodes using
        Label1 of <current_task>, and Label2 of <current_task>,
        and New of <current_task>.
    If Type of <current_task> is split_node, Then
      Accomplish_goal: Split node using
        Label of <current_task>, and New_label1 of <current_task>,
        and New2 of <current_task>, and Child1 of <current_task>,
        and Child2 of <current_task>, and Child3 of <current_task>,
        and Child4 of <current_task>, and Parent1 of <current_task>,
        and Parent2 of <current_task>, and Parent3 of <current_task>,
        and Parent4 of <current_task>.
    If Type of <current_task> is alter_node_description, Then
      Accomplish_goal: Change attribute using
        node, and "Description", and alter,
        and Label of <current_task>, and New of <current_task>.
    If Type of <current_task> is extend_node_description, Then
      Accomplish_goal: Change attribute using
        node, and "Description", and xtend,
        and Label of <current_task>, and Extension of <current_task>.
    If Type of <current_task> is alter_node_reference, Then
      Accomplish_goal: Change attribute using
        node, and "Node␣Reference", and alter,
        and Label of <current_task>, and New of <current_task>.
    If Type of <current_task> is extend_node_reference, Then
      Accomplish_goal: Change attribute using
        node, and "Node␣Reference", and xtend,
        and Label of <current_task>, and Extension of <current_task>.
    If Type of <current_task> is add_edge, Then
      Accomplish_goal: Add edge using
        Source of <current_task>, and Target of <current_task>.
    If Type of <current_task> is alter_edge, Then
      Accomplish_goal: Alter Edge using
        Change of <current_task>, and Source of <current_task>,
        and Target of <current_task>, and New of <current_task>.
    If Type of <current_task> is delete_edge, Then
    Accomplish_goal: Remove edge using
      Source of <current_task>, and Target of <current_task>.
  If Type of <current_task> is add_fact, Then
    Accomplish_goal: Add fact using
      Description of <current_task>.
  If Type of <current_task> is delete_fact, Then
    Accomplish_goal: Remove fact using
      Label of <current_task>.
  If Type of <current_task> is alter_fact_description, Then
    Accomplish_goal: Change attribute using
      fact, and "Description", and alter,
      and Label of <current_task>, and New of <current_task>.
  If Type of <current_task> is extend_fact_description, Then
    Accomplish_goal: Change attribute using
      fact, and "Description", and xtend,
      and Label of <current_task>, and Extension of <current_task>.
  If Type of <current_task> is determine_fact, Then
    Accomplish_goal: Determine fact.
  Return_with_goal_accomplished.

//
    ****************************************************************************
// Graph Operations
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Change Parent:
// change the target of the edges between nodes with the given source labels
// and the target parent to the new parent node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Change parent using
                 <cp_old_parent>, and <cp_new_parent>, and <cp_child1>,
                 and <cp_child2>, and <cp_child3>, and <cp_child4>
  Step 1. Decide:
          If <cp_child1> is nil, Then
            Goto 6;
          If <cp_child2> is nil, Then
            Goto 5;
          If <cp_child3> is nil, Then
            Goto 4;
          If <cp_child4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Alter edge using
                           <cp_child4>, and <cp_old_parent>,
                           and arrow, and <cp_new_parent>.
  Step 3. Accomplish_goal: Alter edge using
                           <cp_child3>, and <cp_old_parent>,
                           and arrow, and <cp_new_parent>.
  Step 4. Accomplish_goal: Alter edge using
                           <cp_child2>, and <cp_old_parent>,
                           and arrow, and <cp_new_parent>.
  Step 5. Accomplish_goal: Alter edge using
                           <cp_child1>, and <cp_old_parent>,
                           and arrow, and <cp_new_parent>.
  Step 6. Return_with_goal_accomplished.
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Change Child:
// change the source of the edges between nodes with the given target labels
// and the source child node to the new child node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Change child using
                 <cc_old_child>, and <cc_new_child>,
                 and <cc_parent1>, and <cc_parent2>,
                 and <cc_parent3>, and <cc_parent4>
  Step 1. Decide:
          If <cc_parent1> is nil, Then
            Goto 6;
          If <cc_parent2> is nil, Then
            Goto 5;
          If <cc_parent3> is nil, Then
            Goto 4;
          If <cc_parent4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Alter edge using
                           <cc_old_child>, and <cc_parent4>,
                           and tail, and <cc_new_child>.
  Step 3. Accomplish_goal: Alter edge using
                           <cc_old_child>, and <cc_parent3>,
                           and tail, and <cc_new_child>.
  Step 4. Accomplish_goal: Alter edge using
                           <cc_old_child>, and <cc_parent2>,
                           and tail, and <cc_new_child>.
  Step 5. Accomplish_goal: Alter edge using
                           <cc_old_child>, and <cc_parent1>,
                           and tail, and <cc_new_child>.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add as Parent:
// add edges directing from the given child nodes to the parent node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add_as parent using
                 <aap_parent>, and <aap_child1>,
                 and <aap_child2>, and <aap_child3>,
                 and <aap_child4>
  Step 1. Decide:
          If <aap_child1> is nil, Then
            Goto 6;
          If <aap_child2> is nil, Then
            Goto 5;
          If <aap_child3> is nil, Then
            Goto 4;
          If <aap_child4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Add edge using
                           <aap_child4>, and <aap_parent>.
  Step 3. Accomplish_goal: Add edge using
                           <aap_child3>, and <aap_parent>.
  Step 4. Accomplish_goal: Add edge using
                           <aap_child2>, and <aap_parent>.
  Step 5. Accomplish_goal: Add edge using
                           <aap_child1>, and <aap_parent>.
  Step 6. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add as Child:
// add edges directing form the child node to the given parent nodes
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add_as child using
                 <aac_child>, and <aac_parent1>,
                 and <aac_parent2>, and <aac_parent3>,
                 and <aac_parent4>
  Step 1. Decide:
          If <aac_parent1> is nil, Then
            Goto 6;
          If <aac_parent2> is nil, Then
            Goto 5;
          If <aac_parent3> is nil, Then
            Goto 4;
          If <aac_parent4> is nil, Then
            Goto 3.
  Step 2. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent4>.
  Step 3. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent3>.
  Step 4. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent2>.
  Step 5. Accomplish_goal: Add edge using
                           <aac_child>, and <aac_parent1>.
  Step 6. Return_with_goal_accomplished.

//
    ****************************************************************************

// Node Operations
//
    ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Node / Necessary Causal Factor (NCF):
// drag fact to parent
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add node using
                 <an_fact>, <an_ptype>, and <an_parent>
  Step 1. Accomplish_goal: Drag item using
                           fact, and <an_fact>, and nil,
                           and <an_ptype>, and <an_parent>, and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Remove Node:
// drag node to trash
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Remove node using
                 <rm_label>
  Step 1. Accomplish_goal: Drag item using
                           node, and <rm_label>, and nil,
                           and trash, and "Trash", and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Insert Node: (between parent node and one or more optional child nodes)
// drag fact to the edge between parent and child node
// alter edges between parent node and given child nodes: new parent is new
    node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Insert node using
                 <in_parent>, and <in_label>,
                 and <in_child1>, and <in_child2>,
                 and <in_child3>, and <in_child4>
  Step 1. Accomplish_goal: Drag item using
                           node, and <in_label>, and nil,
                           and edge, and <in_child1>, and <in_parent>.
  Step 2. Accomplish_goal: Change parent using
                           <in_parent>, and <in_label>,
                           and <in_child2>, and <in_child3>,
                           and <in_child4>, and nil.
  Step 3. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Merge Nodes: (take care of corresponding visual objects!)
// select the nodes to merge field
// drag selection to merge field
// change the node description
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Merge nodes using
                 <mn_label1>, and <mn_label2>, and <mn_new_label>
  Step 1. Accomplish_goal: Select items.
  Step 2. Accomplish_goal: Drag item using
                           selection, and nil, and nil,
                           and field, and "Merge", and nil.
  Step 3. Accomplish_goal: Change attributen using
                           node, and "Description",
                           and xtend, and <mn_label1>,
                           and <new_label>.
  Step 4. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Split Node: (take care of corresponding visual objects!)
// drag node to split to split field
// type in the label of the new sibling node
// change the source node description
// add edges between given child nodes: new parent is new node
// add edges between given parent nodes: new child is new node
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     *

Method_for_goal: Split node using
                 <sn_label>, and <sn_new_label1>, and <sn_new_label2>,
                 and <sn_child1>, and <sn_child2>, and <sn_child3>,
                 and <sn_child4>, and <sn_parent1>, and <sn_parent2>,
                 and <sn_parent3>, and <sn_parent4>
  Step 1. Accomplish_goal: Drag item using
                           node, and <sn_label>, and nil,
                           and field, and "Split", and nil.
  Step 2. Type_in <sn_new_label1>.
  Step 3. Accomplish_goal: Change attribute using
                           node, and "Description",
                           and alter, and "New␣Node",
                           and <sn_new_label2>.
  Step 4. Accomplish_goal: Add_as child using
                           <sn_new_label2>, and <sn_parent1>,
                           and <sn_parent2>, and <sn_parent3>,
                           and <sn_parent4>.
  Step 5. Accomplish_goal: Add_as parent using
                           <sn_new_label2>, and <sn_child1>,
                           and <sn_child2>, and <sn_child3>,
                           and <sn_child4>.
  Step 6. Return_with_goal_accomplished.

//
     ****************************************************************************
// Attribute Operators
//
     ****************************************************************************


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     *
// Change Attribute:
// click node or fact to be changed
// if the description is not to be changed, select the corresponding input
     field
// if the txt shall be replaced, type delete key
// enter the node description
// confirm the change by hitting the 'return'-key
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     *

Method_for_goal: Change attribute using
                 <ct_type>,and <ct_attribute>, and <ct_a_or_x>,
                 and <ct_original>, and <ct_new>
  Step 1. Accomplish_goal: Click_at item using
                           <ct_type>, and <ct_original>, and nil.
  Step 2. Decide:
          If <ct_attribute> is_not "Description", Then
            Accomplish_goal: Click_at item using
                             input_field, and <ct_attribute>, and nil.
  Step 3. Decide:
          If <ct_a_or_x> is alter, Then
            Keystroke DEL.
  Step 4. Type_in <ct_new_label>.
  Step 5. Keystroke CR.
  Step 6. Return_with_goal_accomplished.
```

```
//
    ************************************************************************
// Edge Operations
//
    ************************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Edge:
// drag from the source node to the targetnode
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add edge using
                 <ae_source>, and <ae_target>
  Step 1. Accomplish_goal: Drag item using
                           node, and <ae_source>, and nil,
                           and node, and <ae_target>, and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Alter Edge:
// drag the edge to the new child / parent
// (method dos not distinguish the change of source or target of an edge
    because
// of the way glean3 implements fitt's law)
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Alter edge using
                 <ale_source>, and <ale_target>,
                 and <ale_end>, and <ale_new>
  Step 1. Accomplish_goal: Drag item using
                           edge, and <ale_source>, and <ale_target>,
                           and node, and <ale_new>, and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Remove Edge:
// drag edge to trash
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Remove edge using
                 <rme_source>, and <rme_target>
  Step 1. Accomplish_goal: Drag item using
                           edge, and <rme_source>, and <rme_target>,
                           and trash, and "Trash", and nil.
  Step 2. Return_with_goal_accomplished.

//
    ************************************************************************
// List of Facts Operations
//
```

```
    ************************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Add Fact:
// select no fact by clicking the canvas
// enter the fact description
// type return key
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Add fact using
                 <af_description>
  Step 1. Accomplish_goal: Click_at item using list, and "List␣of␣Facts", nil.
  Step 2. Type_in <af_description>.
  Step 3. Keystroke CR.
  Step 4. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Remove Fact:
// drag fact to trash
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Remove fact using
                 <df_description>
  Step 1. Accomplish_goal: Drag item using
                           fact, and <df_description>, and nil,
                           and trash, and "Trash", and nil.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Determine Fact:
// find next fact (dummy: a verify operator which consumes search/decision time
    )
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Determine fact
  Step 1. Verify "Find␣the␣fact␣that␣is␣causal␣factor".
  Step 2. Return_with_goal_accomplished.

//
    ************************************************************************
// Basic Operations
//
    ************************************************************************

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Look for Item:
// find the item
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
```

```
Method_for_goal: Look_for item using
                 <lfi_type>, and <lfi_spec1>, and <lfi_spec2>
  Step 1. Decide:
          If <lfi_type> is edge, Then
            Look_for_object_whose Type is <lfi_type>,
            and Source is <lfi_spec1>, and Target is <lfi_spec2>
            and_store_under <object>;
          Else
            Look_for_object_whose Type is <lfi_type>, and Label is <lfi_spec1>
            and_store_under <object>.
  Step 2. Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Click at Item:
// find the item
// point to the item
// click the mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Click_at item using
                 <cai_type>, and <cai_spec1>, and <cai_spec2>
  Step 1. Accomplish_goal: Look_for item using <cai_type>, and <cai_spec1>, and
      <cai_spec2>.
  Step 2. Point_to <object>.
  Step 3. Click mouse_button.
  Step 4. Delete <object>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Drag item:
// find the item
// point to the item
// hold down the mouse button
// find the target
// point to the target
// release the mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *

Method_for_goal: Drag item using
                 <di_stype>, and <di_sspec1>, and <di_sspec2>,
                 and <di_ttype>, and <di_tspec1>, and <di_tspec2>
  Step 1. Accomplish_goal: Look_for item using <di_stype>, and <di_sspec1>, and
      <di_sspec2>.
  Step 2. Point_to <object>.
  Step 3. Hold_down mouse_button.
  Step 4. Accomplish_goal: Look_for item using <di_ttype>, and <di_tspec1>, and
      <di_tspec2>.
  Step 5. Point_to <object>.
  Step 6. Release mouse_button.
  Step 7. Delete <object>; Return_with_goal_accomplished.

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
// Select items:
// (click & drag on the canvas, where to start and to stop is not of importance
    ,
// because the fitt's law implementation)
// point to the canvas
// hold down mouse button
// move mouse (pointing again to the canvas)
// release mouse button
// finished.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  *

Method_for_goal: Select items
  Step 1. Look_for_object_whose Type is canvas and_store_under <si_object>.
  Step 2. Point_to <si_object>.
  Step 3. Hold_down mouse_button.
  Step 4. Look_for_object_whose Label is canvas and_store_under <si_object>.
  Step 5. Point_to <si_object>.
  Step 6. Release mouse_button.
  Step 7. Delete <si_object>; Return_with_goal_accomplished.
```

Bibliography

[Annett 67]	John Annett & Keith Duncan. *Task Analysis and Training Design*. Occupational Psychology, vol. 41, pages 211–221, 1967.
[Baumeister 00]	Lynn K. Baumeister, Bonnie E. John & Michael D. Byrne. *A comparison of tools for building GOMS models*. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 502–509. ACM Press, 2000.
[Card 80]	Stuart K. Card, Thomas P. Moran & Allen Newell. *The keystroke-level model for user performance time with interactive systems*. Communications of the ACM, vol. 23, no. 7, pages 396–410, 1980.
[Card 83]	Stuart K. Card, Allen Newell & Thomas P. Moran. The psychology of human-computer interaction. Lawrence Erlbaum Associates, Inc., 1983.
[Cooper 95]	Alan Cooper. About face: The essentials of user interface design. IDG Books Worldwide, Inc., Foster City, CA, USA, 1995.
[Dix 98]	Alan Dix, Janet Finlay, Gregory Abowd & Russell Beale. Human computer interaction (second edition). Prentice Hall, 1998.
[Freed 03]	Michael A. Freed, Michael Matessa, Roger Remington & Alonso Vera. *How Apex Automates CPM-GOMS*. In Fifth International Conference on Cognitive Modeling, April 2003.
[Freed 04]	Michael A. Freed. *Apex Homepage: Usable Autonomy*. http://human-factors.arc.nasa.gov/apex/index.html, 2004.
[Frohlich 97]	David M. Frohlich. *Direct Manipulation and other Lessons*. In Handbook of Human Computer Interaction. Elsevier, 1997.
[Gray 92]	Wayne D. Gray, Bonnie E. John & Michael E. Atwood. *The precis of Project Ernestine or an overview of a validation of GOMS*. In

CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 307–312. ACM Press, 1992.

[Hackos 98] JoAnn T. Hackos & Janice C. Redish. User and task analysis for interface design. John Wiley & Sons, Inc., New York, New York, USA, 1998.

[Hennig 03] Jan E. Hennig. Konzeption eines verteilten Datenarchivierungssystems. Master's thesis, Universität Bielefeld, http://www.rvs.uni-bielefeld.de, RVS-Dip-03-01, September 2003.

[John 88] Bonnie E. John. *Contributions to Engineering Models of human-computer interaction*. PhD thesis, Carnegie Mellon University, 1988.

[John 95a] Bonnie John. *Why GOMS?* Interactions, vol. 2, no. 4, pages 80–89, 1995.

[John 95b] Bonnie E. John & Wayne D. Gray. *CPM-GOMS: an analysis method for tasks with parallel activities*. In CHI '95: Conference companion on Human factors in computing systems, pages 393–394. ACM Press, 1995.

[John 96] Bonnie E. John & David E. Kieras. *Using GOMS for user interface design and evaluation: which technique?* ACM Transactions on Computer-Human Interaction, vol. 3, no. 4, pages 287–319, 1996.

[John 02] Bonnie John, Alonso Vera, Michael Matessa, Michael Freed & Roger Remington. *Automating CPM-GOMS*. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 147–154. ACM Press, 2002.

[Jonassen 89] David H. Jonassen, Wallace H. Hannum & Martin Tessmer. Handbook of task analysis procedures. Praeger Publishers, Westport, CT, USA, 1989.

[Kandel 00] Eric R. Kandel, James H. Schwartz & Thomas M. Jessel. Principles of neural science. Appleton & Lange, 4. edition, 2000.

[Kieras 85] David E. Kieras & Peter G. Polson. *An approach to the formal analysis of user complexity*. International Journal of Man-Machine Studies, vol. 22, pages 365–394, 1985.

[Kieras 88] David E. Kieras. *Towards a Practical GOMS Model Methodology for User Interface Design*. In Handbook of Human-Computer Interaction. North-Holland, New York, NY, 1988.

[Kieras 95] David E. Kieras, Scott D. Wood, Kasem Abotel & Anthony Hornof. *GLEAN: a computer-based tool for rapid GOMS model usability evaluation of user interface designs*. In Proceedings of the 8th annual ACM symposium on User interface and software technology, pages 91–100. ACM Press, 1995.

[Kieras 96] David E. Kieras. *A Guide to GOMS Model Usability Evaluation using NGOMSL*, 1996.

[Kieras 97a] David E. Kieras. *GOMS Model Usability Evaluation Using NGOMSL*. In Handbook of Human Computer Interaction. Elsevier, 1997.

[Kieras 97b] David E. Kieras & David E. Meyer. *An Overview of the EPIC Architecture for Cognition and Performance With Application to Human-Computer Interaction*. Human-Computer Interaction, vol. 12, no. 4, pages 391–438, 1997.

[Kieras 99] David E. Kieras. *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3*. University of Michigan, January 1999.

[Kieras 04] David E. Kieras & Thomas P. Santoro. *Computational GOMS modeling of a complex team task: lessons learned*. In Proceedings of the 2004 conference on Human factors in computing systems, pages 97–104. ACM Press, 2004.

[Kieras 05] David E. Kieras. *GOMS Models: An Approach to Rapid Usability Evaluation*. http://www.eecs.umich.edu/˜kieras/goms.html, 2005.

[Kirwan 92] Barry Kirwan & Les K. Ainsworth. A guide to task analysis. Taylor & Francis Ltd., London, England, 1992.

[Ladkin 99] Peter B. Ladkin. *A Quick Introduction to Why-Because Analysis*. http://www.rvs.uni-bielefeld.de, March 1999.

[Ladkin 01] Peter B. Ladkin & Karsten Loer. Why-Because Analysis: Formal reasoning about incidents. http://www.rvs.uni-bielefeld.de, RVS-Bk-01-01, 2001.

[Ladkin 03] Peter B. Ladkin, Lars Heidiecker, Nils Hoffmann, Peter Husemann, Jan Paller, Jan Sanders, Jörn Stuphorn & Andreas Vangerow. *WBA of the Royal Majesty Accident*. http://www.rvs.uni-bielefeld.de, RVS-RR-03-01, July 2003.

[Ladkin 05] Peter B. Ladkin. *Networks and distributed Systems Homepage*. http://www.rvs.uni-bielefeld.de, 2005.

[Loer 98] Karsten Loer. Towards "Why...Because"-Analysis of failures. Master's thesis, Univerität Bielefeld, http://www.rvs.uni-bielefeld.de, RVS-Dip-98-02, Februar 1998.

[Macaulay 95] Linda Macaulay. Human computer interaction for software designers. Internat. Thomson Computer Press, 1995.

[Norman 88] Donald A. Norman. The design of everyday things. Doubleday Press, New York, 1988.

[Raskin 00] Jef Raskin. Humane interface: New directions for designing interactive systems. Addison-Wesley Publishing, 2000.

[Schulmeister 97] Rolf Schulmeister. Grundlagen hypermedialer Lernsysteme. Theorie - Didaktik - Design. Oldenbourg Verlag, Muenchen, 2. edition, 1997.

[Shneiderman 82] Ben Shneiderman. *The Future of Interactive Systems and the Emergence of Direct Manipulation*. Behaviour and Information Technology, vol. 1, no. 3, pages 237–256, 1982.

[Sieker 04] Bernd Sieker. Visualisation concepts and improved software tools for Causal System Analysis. Master's thesis, Universität Bielefeld, http://www.rvs.uni-bielefeld.de, RVS-Dip-04-01, February 2004.