UNIVERSITÀ DEGLI STUDI DI ROMA "LA SAPIENZA"
DOTTORATO DI RICERCA IN INFORMATICA

Modelling Interactive Computing exploiting the Undo

Roberta Mancini

IX-96-6

Roberta Mancini

Modelling Interactive Computing exploiting the Undo

Dottorato di Ricerca in Informatica

IX-96-6

Università degli Studi di Roma "La Sapienza"

AUTHOR'S ADDRESS:
ROBERTA MANCINI
DIPARTIMENTO DI SCIENZE DELL'INFORMAZIONE
UNIVERSITÀ DEGLI STUDI DI ROMA "LA SAPIENZA"
VIA SALARIA 113, 00198 ROMA, ITALY
E-MAIL: ROBERTA@DSI.UNIROMA1.IT

To my mother, Anna

Acknowledgments

I am sincerely grateful to my advisor Prof. Stefano Levialdi, which introduced me to the HCI world. He believed in my scientific ability since I was an undergraduated student. Following his teaching, I learnt how to do research.

Many thanks are due to Prof. Mirella Casini Schaerf and Prof. Rocco De Nicola which, members of the internal thesis committee, provided me useful comments and suggestions.

I want to thank all my friends and colleagues in Huddersfield (UK), at the School of Computing and Mathematics, and in Rome, at the Department of Computer Science, University of Rome, "La Sapienza". I spent with them many enjoyable moments and they supported me with their frienship, useful suggestions and real help.

I am grateful to the Dix family, for their welcome and friendship. In particular Alan, without which this thesis could not have been possible. Alan has been, and is, a teacher, a guide, a friend. He taught me not simlpy how to use formal methods, but to really "feel" them.

A special thought is for my family, for the love, patience, comprehension and help that they give me.

Finally, I want to thank God for all He gave me.

Introduction

Human-Computer Interaction (HCI) is a multi-disciplinary area, since it involves computer science, psychology, ergonomics, ... In any of these fields, research has been done in order to understand and improve the interaction, making it as easy and fruitful as possible. Due to its nature, there is not a common understanding of an "interactive system", because a definition, eventually provided by psychologists, would not take into account all the aspects relevant to a computer scientist, and vice versa. In this work, we are going to introduce interactive systems as a subset of the reactive ones [69] and then describe them in terms of their basic features: modularization, feedback, dialogue, usability and others. One of such feature is the *Undo* function, which allows the user to cancel the effect of his past action so reaching a previous state. Interactivity is not based only on the presence of *Undo*, yet no real interactive system may live without it. Moreover, during the requirements analysis, which is the first step in the softfware life cycle [32], usually, even users which are not computer experts, require a function which allows them to repair an error. So *Undo* is not simply a feature of interactive systems, but it is a real user need.

Someone could argue that *Undo* is a function typical not only for interactive systems, but also for traditional ones. Some people consider 'any recovery function a kind of undo'. The fact that we disagree with this belief, may be due the lack of a common understanding on the *Undo*. Moreover, many kinds of *Undo* exists. Due to the relevance of this function in interactive systems, we think that a better understanding is important, because the knowledge of this function and of its potentiality gives strongly contribution to improve interaction making it more confident and fruitful.

The aim of this thesis is to clarify the scope of the *Undo* function. Such a clarification is done at three levels. The first level is inside the world of reactive systems and recovery functions, in order to place the *Undo* function on the map of reactive systems. The second level explores different kinds of

Undo with which a user can interact. The third level analyses in depth a specific class of *Undo*, the backtrack *Undo*. On the first two levels, we use a descriptive analysis in order to clarify what exactly *Undo* means, when it can be used, on which actions it operates, ..., what do we intend by *Redo*, what is the link between *Undo* and *Redo*, etc. Conversely, in the third and last level, we accomplish a formal analysis of a specific undo mechanism: the backtrack *Undo*. By applying the PIE model [27], we formally describe the behaviour of this class of systems and, after introducing the concept of conservative encapsulation and of behavioural equivalence, we prove some interesting and useful properties.

Depending on the interaction level, we may consider the set of reactive systems in Computer Science as organised in ordinary reactive systems, operating systems and application software. In ordinary reactive systems there is no interaction, but simply reaction between the entities involved in the communication. In both the other two cases, while working with operating systems and application software, interaction is present, since there is a continuous exchange of information between user and system. Moreover, there is also an immediate computer reaction to any user action, reaction which may be more or less explicitely represented to the user by visual and/or audio feedback, depending on the nature of the interaction language. The difference between the interaction with an operating system and an application software is in the object of interest of any user action: within operating systems the object of interest is a file, within application software such object is the file content.

We call any functions which allows the user to reach a past state a *recovery function*. Morover, since, as just above mentioned, there is more than one kind of reactive system, we can have different kinds of recovery functions, depending on the reactive system itself. We have *ordinary recovery functions* with reactive systems; *implicit undo* at the file level (implicit in the sense that a system function called *Undo* is not available, but it is possible to reach a past state using some other system functions and/or by direct manipulation); *explicit undo* at the application level. At this point we know that we could expect to have an explicit undo only available at the application level.

Undo is a very important system function, not only because it permits the user to repair an error, if it occurs, by cancelling the last performed action(s) (the plural indicates that *Undo* may cancel more than one action, depending on the implementation), but also because it allows the user to *handle* and

reduce non-determinism in interaction. In HCI with non-determinism we mean the impossibility to perform a prediction on an object, impossibility due to the incomplete knowledge on the object itself. We have a double non-determinism in interaction: one is from the system point of view, because it is not possible to foresee the human behaviour; the other is from the user point of view, because if the user has not a full knowledge on the system states and behaviour, his predictions may differ from what will really happen. When the user has not enough information on what he can do with the system, he combine the visual approach with direct manipulation and navigate through the system itself. In this case, the use of the *Undo* function, which allows to reach a past state, enables him to try different paths while interacting. Consequently, the user increases his knowledge on the system potentiality, i.e. he can handle non-determinism. Moreover, when a user performs a command reaching a different state from the one he foresaw, then there is some inconsistency between the user and system points of view.

By performing *Undo*, the user can not only repair an error, but can also resolve such inconsistencies and can increase his knowledge on the computer behaviour, so reducing non-determinism. Unfortunately, *Undo* may also add non-determinism in interaction, and this happens when it is not available when we think it is, or when its effect is different from what we foresaw, that is when the user and system models of the *Undo* differ. Basically, such a difference is due to the fact that for the user, *Undo* is a command, and, as all other ordinary commands, should act modifying the content of a file. Actually, *Undo* is a meta-command, since its domain of interest is the action history. We can have two classes of *Undo* systems, one for which it is self-applicable, that is *Undo* is forced to belong to its domain of interest, and its effect may be cancelled by performing another *Undo*, or one for which it is not self-applicable, it is not possible to undo the effect of an *Undo* , enabiling it to be used as a backtrack tool.

To order the different kinds of *Undo* , we provide an informal definition of *Undo* and a taxonomy of interactive systems, the last based on the *Undo* mechanisms: we consider the repetition of *Undo* (if *Undo* of *Undo* is or is not allowed) and the granularity (possibility to undo only the last past action or a block of past actions). Next, we introduce the *Redo* function, which is not simply the inverse of *Undo* . After providing also an informal definition for *Redo*, we explain the link and the dependencies between *Undo* and *Redo* and we propose another taxonomy of interactive systems, this time also including the just introduced function.

Taxonomies are important to classify the different kinds of *Undo*, but they are not able to describe properties of different *Undo* mechanisms. To this aim, we need a formal approach. Among the different formal approaches, we use a black-box model, the PIE model, because we are interested in describing properties of the system behaviour from the user point of view, and to this aim a black box model seems to be the most suitable. Using the PIE model, we describe the behavior of the backtrack *Undo*. We next provide a definition of *conservative encapsulation*, which expresses the idea that given a system without *Undo* , and the augmented system enriched by *Undo*, the old system is still inside the augmented one.

We introduce a definition of behavioural equivalence between systems, then proving that all systems $\mathcal{P}^b$ which are the backtrack *Undo* of the same original PIE $\mathcal{P}$ are behaviourally equivalent. In such proof, we do not involve the set of states, since the equivalence is only in terms of the effective history. Nevertheless, we can say something also on the set of states; in fact, we define two particular PIEs, $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$, which are the backtrack *Undo* of the same original PIE $\mathcal{P}$, with the "bigger" and "smallest" sets of states respectively. We prove that for any backtrack *Undo* $\mathcal{P}^b$ of the same original system $\mathcal{P}$, there exists a homomorphism from the set of states of $\mathcal{P}^{max}$ to the one of $\mathcal{P}^b$, and similarly, we prove that for any backtrack *Undo* $\mathcal{P}^b$ of the same original system $\mathcal{P}$, there exists a homomorphism from the set of states of $\mathcal{P}^b$ to the one of $\mathcal{P}^{min}$. The meaning of the two theorems on the homomorphisms is that, even if we do not have enough information on the set of states, in some sense we can find an upper bound and a lower bound. Moreover, we prove that the class of all the backtrack *Undo* of the same original system $\mathcal{P}$ is a category, whose initial and terminal element are $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ respectively. This means that not only a 0-morphism between $\mathcal{P}^{max}$ and $\mathcal{P}^b$, and similarly between $\mathcal{P}^b$ and $\mathcal{P}^{min}$ exists, but also that such 0-morphisms are unique. This means that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the lowest upper bound and greatest lower bound, that is we cannot have more information than $\mathcal{P}^{max}$ neither less then $\mathcal{P}^{min}$. Since the maximal system allows the user to know precisely the actual state and how he reached it, then he can reduce non-determinism and, consequently, can feel more in control of his dialogue while interacting. For this reason, during the development of an application software, before to choose which kind of *Undo* has to be implemented, developers should take in to account also analysis provided by applying formal approaches, in order to increase the usability of the application software allowing an easy and fruitful interaction.

This thesis is organised as follows. In Chapter 1 we introduce HCI as a

multi-disciplinary area and the corresponding formal methods are discussed, with particular refernce to the PIE model. In Chapter 2 we introduce reactive and interactive systems and we describe properties of interaction languages. In Chapter 3 we discuss different kinds of recovery function, while in Chapter 4 we characterise non-determinism in interaction. Different kinds of *Undo* mechanisms are analysed in Chapter 5 and two taxonomies of interactive systems, based on the different kinds of *Undo* and *Redo* are proposed. In Chapter 6, using the PIE model, we highlight some formal properties of the class of backtrack *Undo* of the same original system $\mathcal{P}$, and we prove that all the backtrack *Undo* of the same original system $\mathcal{P}$ are behaviourally equivalent. In Chapter 7 we introduce two PIE, $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ (having the "biggest" and "smallest" set of states respectively) and we prove the theorem of homomorphisms from the set of states of $\mathcal{P}^{max}$ to the one of $\mathcal{P}^{b}$, and from the set of states of $\mathcal{P}^{b}$ to the one pf $\mathcal{P}^{min}$. Moreover, we prove that the class of all the backtrack *Undo* $\mathcal{P}^{b}$ of the same original PIE $\mathcal{P}$ is a category of which $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the initial and terminal elements. The main results of this work and possible extension are described in the conclusion.

Chapter 1

Human-Computer Interaction

Human-Computer Interaction (HCI) is a multidisciplinary area, whose aim is to understand all the aspects related to the communication between a human and a computer in order to provide suggestions for designing usable interactive systems. Since the two involved partners in the communication are a human and a computer, the aspects of such a communication may be mainly analised from two different points of view, from the psychology and the computer science point of view. However, neither analysis from psychology nor computer science is complete, since the suggestions provided by the phychology are informal and sometimes may not cover all the system functions, while suggestions provided by computer scientists may not consider all the human or contextual aspects. For this reason, a combination of contributions from these two main research areas is important in order to provide guidelines for developing usable interactive systems.

In this Chapter, after an introduction on the origins of Human-Computer Interaction, a psychological point of view on interaction is given, focusing particularly on interfaces and on usability aspects. Next, formal methods in interaction are introduced, giving attention to formal models for interactive systems. In particular, we discuss formal approaches to dialogue specification and analysis, specification of individual interactive systems and generic models of interactive systems. Among the generic models, emphasis is given to the PIE model [27] and to some of its basic properties, which will be used in the following Chapters.

1.1 The origins of Human-Computer Interaction

Since the beginning of the history of computing there has been the need to improve the communication between a user and a machine, trying to make such a communication, depending on the available technology, as easy and comfortable as possible.

One of the first references to an easy and fruitful utilization by a human of a computer is by Vannevar Bush and dates back to 1945 [16]. In his paper "As we may think", he suggested the use of a device called MEMEX:

> ... A MEMEX is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility...

He described how the system should function, the way in which information could be retrieved, a browser mechanism for books, different indexing techniques, ... His description foresaw a multimedia system for an application of information storage and retrieval. Such a description represented the first expression for the need of a mechanical tool, to support human work, fast and easy to be employed by a user. Actually, similar systems exist but, considering that the first computers appeared in the Fifties, in 1945 Bush's opinion looked as science fiction.

Some years later, in 1960, another view of the importance of the human-computer interaction has been offered by Licklider [48]. He referred to the human and the computer as two entities that should work in symbiosis. But the other important thing that he highlighted was the fact that a computer should be used not only to solve preformulated problems, but also to solve problems which could arise during the computation, introducing for the first time the idea of a partial result:

> ... Present-day computers are designed primarily to solve preformulated problems or to process data according to predetermined procedures. The course of the computation may be conditional upon results obtained during the computation, but all the alternatives must be foreseen in advance...

The main computer features that, at the beginning of the Sixties, the users required, were the time-sharing of computers among many users and an electronic input-output surface. It was and is extremely important to

see what a user asks to a computer (input) and what the computer itself provides as answer (output).

Going on with the years, the number of features that a human required from the computer increased. This is due to the fact that around the Seventies, with the advent of mainframes and minicomputers, an increasing number of non-programmer users have been forced to insume a strong effort in using manuals and handbooks before being able to employ a computer. So good human-computer interaction started to be a real necessity, while in the Fifties-Sixties it was a proposal, a possibility.

The Eighties, with the intoduction of direct manipulation paradigm [81], represented really a revolution in human-computer interaction. By employing direct manipulation, the user is not forced any more to learn, by consulting manuals and handbooks, how to employ the system, but he can do it by simply navigating through it. Moreover, in order to allow the user to feel more confident while interacting, different recovery functions are generally available, so that he can repair an error if it occurs. With the advent of direct manipulation, the user, his aims, goals, needs, etc, started to have more and more importance along the development of application software and human-computer interaction became a real and wide research area.

1.2 HCI as a multi-disciplinary area

The aim of the Human-Computer Interaction (HCI) [53, 32, 50, 73] is to understand the interaction between a human and a computer in all its aspects and to provide suggestions and guidance for designing computer systems which are really usable, that is which allow humans to accomplish their tasks in a very easy and fruitful way. The involved partners in the communication in HCI are two: a human and a computer. For this reason, in order to express properties that an interactive system should have to increase its usability [5, 80, 63], we need to take into account characteristics of both partners. In fact, systems developed only taking into account algorithmic, computational, implementative aspects are often not usable; conversely, the design of a system provided by a user could be not feasible. For this reason, psychology and computer science must work together. However, also other disciplines play a very important role in HCI, as linguistics, artificial intelligence, ergonomics, etc [6]. For this reason, HCI is a multi-disciplinary area.

1.3 Interaction models

The essence of the interaction between a human and a computer is in their continuous exchange of information: the user establishes a goal, then provides an input, the computer provides an answer to the given input, the user evaluates the feedback and then he can provide another input. Since the computer and the human have very different characteristics, it is important to find a common ground in which they can communicate. Such a ground is represented by the interface, through which the translation from the "human world" to the computer one (and vice versa) takes place. Anyway, such a translation can fail for different reasons, and the use of some interaction models can help us in understanding which are the main difficulties, so that we can find a way to solve them.

Different interaction models have been proposed [32]. One of the most influential in HCI has been the Norman's model, perhaps because it is very near to the intuitive idea of the essence of human-computer interaction. In this model, the cycle of user-input/computer-answer is analised only from the user point of view, and can be seen as composed of two main phases: execution and evaluation. These may be refined into the seven phases listed below [65]:

(1) establishing the goal;

(2) forming the intention;

(3) specifying the action sequence;

(4) executing the action;

(5) perceiving the system state;

(6) interpreting the system state;

(7) evaluating the system state with respect to the goals and intentions.

The first three phases correspond to the understanding on what the user has to do in order to reach his aim, that is the understanding on which task he has to perform and how to accomplish it by employing the computer. The fourth phase is the execution of the established action, while the fifth and sixth phases correspond to the perception and understanding of the computer reaction to the previous user action. Finally, the last phase

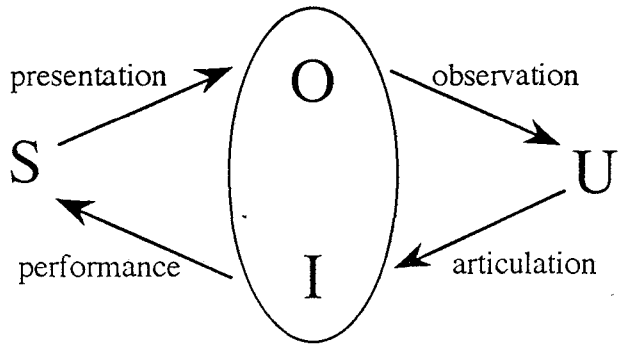

Figure 1.1: The general interaction framework.

corresponds to the evaluation of the reached state with respect to the user aim: if the reached state does not reflect the user aim, then he can formulate another goal and repeat the cycle. This final "evaluation" phase suggests the possibility of errors occurring, being deteced and corrected. This detection and recovery from error as part of interaction is one of the reasons why most interactive computer systems now support some form of undo. We will return to this issue in succeding chapters.

As said before, the Norman's model analises the interaction cycle only from the user point of view, whereas the computer is simply considered as the interface. Another interaction model, explicitly involving the computer, is the general interaction framework, proposed by Abowd and Beale [2].

In this model, represented in Figure 1.1, we have four components, the system (S), the user (U), the input (I) and the output (O). After the user has identified his goal and what he has to do in order to reach it, then he must provide his input to the computer. To do this, he needs to articulate his request in the input language. In the following step, the input is translated into the system language, and the system executes the required action, which, at this point is presented to the user as output. The user observes the result and, evaluating it with respect to his aim, can decide to eventually start again the interaction cycle. In this model, the input and output components meet on the interface, which is the communication channel between a human and a computer. Such a communication is bidirectional: one direction is from the human to the computer (the input), the other is from the computer to the human (the output).

1.4 The interface

The user interface is the part of a system the users interact with. It is hard to describe good user interfaces by words without showing them, and it is even harder to write about something that is interactive without presenting it interactively. But, then, what is a good interface? The best ones are those that a person cannot see, i.e. are transparent, so there is no interference between the user and the task, e.g. writing on paper [61]. There are different ways in which a human and a computer may communicate, depending on the different interfaces, i.e. on the interaction styles. In the taxonomy on man-computer interaction by Ben Shneiderman [81], the advantages and disadvantages of five interaction styles are illustrated: menu selection, form fill-in, command language, natural language and direct manipulation. Visual and graphic approaches are powerful representation of the 'world of action' [34], which include selectable displays of objects and operations. Then, by pointing, zooming and panning these objects, the user can rapidly perform actions, immediately see their results and, if needed, reverse the last action (undo).

In computer science applications, interface design is taking more and more into account the results of psychological research. Neuro-physiological processes, such as brain reaction to different visual patterns or different responses to color and brightness, contribute to attention focusing and in this way heavily influence knowledge acquisition conditions, when exploiting visual learning. This leads to a new way of interpreting human interaction as a whole, and in particular to the need to take into account human behaviour and responses for an adequate design of computer interfaces. The focus of this design has moved from an application-centered approach, often too difficult to be understood by users, to a more user-centered perspective. Following a user-centered approach, the users are involved in different ways during the development of an application software, e.g. as part of the design team, subjects in user analysis studies, as main actors in simulation sessions and in evaluation activities, they can also comment on working versions of systems.

In order to evaluate how "good" a user interface is, several aspects need to be taken into account, because a user interface is not composed only of separate hardware and software elements. Therefore, its quality depends partially on the hardware (e.g. how display resolution or processor speed impact on the decisions), partially on its functionality (e.g. whether the user goals can be reached with the system) and partially on its usability

(e.g. how easily users can accomplish their tasks).

Recently, emphasis has been posed on evaluation techniques, since there are several reasons to evaluate systems. People involved in developing software products are interested in evaluations to assist in making design decisions and to determine whether or not the products achieve the quality measure that must be met. Cognitive psychologists have been interested in evaluating software in order to study general aspects of human cognition. Individuals, who buy the software because they want to use it, need to evaluate the software before purchasing it to check whether it answers key questions about usability. Even if there is some similarity in the various approaches, since they are all trying to answer whether the system adequately meets the needs of the user, the above interests are not served by a single evaluation methodology. For this reason, a variety of techniques have been proposed [22].

1.5 Usability aspects

The concept of usability, very difficult to grasp in a single, natural definition, has been expressed in the literature with a variety of schemes different from each other. Some of the most significant will be mentioned below. Beginning in 1971, Miller [58] gave his definition of usability in terms of ease of use. Much later, in 1991, Shackel [80] gave another definition, an operational one, considering four factors:

(a) effectiveness, defined as the requested range of tasks that must be carried out at better than a determined performance level; those tasks must be performed by a required percentage of a specified range of users;

(b) learnability, which refers to the performance achieved after some specified time, using some specified amount of training and support;

(c) flexibility, defined as adaptation to some specified range of variation in user tasks;

(d) attitude, which refers to acceptable levels of human cost (e.g. fatigue, discomfort) and perceived benefits.

More recently, in 1993, Badre [5] defined usability of a software product as ease of learning and ease of use: this means that less effort should be

required to the user to perform a task and also that he should make less errors. In the last years usability has became one of the major research areas in HCI, so that the need arose to provide a standard definition. Two accepted definitions are:

- ISO 9126 [41] (software product evaluation): "A set of attributes of software which bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users".
- ISO 9241 Part 11 [42] (ergonomics requirements for office work with Video Terminals): "The effectiveness, efficiency and satisfaction with which specified users can achieve specific goals in a particular environment".

MUSiC (Metrics for Usability Standars In Computing), an Esprit Project (Esprit Project #5429) [54], has provided some tools to measure its quantitative components, i.e.the measurable elements of the quality of interaction between the user and the overall system. Its definition is "the extent to which a product can be used with efficiency, effectiveness and satisfaction by specific users to achieve specific goals in specific environments" [51]. As we can see, this definition is in accordance to the reccomandation of the ISO 9241- 11 (see above). Bevan and Macleod say in [52] that the quality of interaction can be measured taking into account three factors:

(1) the effectiveness, i.e. the extent to which the established goals of the system can be achieved;

(2) the efficiency, i.e. the resources as time, money, mental effort spent to achieve the established goals;

(3) the satisfaction, i.e. the degree of acceptance of the overall system.

They also emphasize that the measures of the three above mentioned quantitative elements do not depend only from the product characteristics, but they are also a function of the context in which the product is to be used. Their scheme, reported in Figure 1.2 (from [52], page 7), illustrates the components of usability. This scheme clarifies that the usability has, as main components, three quantitative elements that make up the quality of interaction, but these ones are tightly binded to the overall system components, i.e. the elements that compose the context. These are:

(1) the users, in fact (as we said in the previous sections) it is fundamental to know the real user, his knowledge, experience and needs;

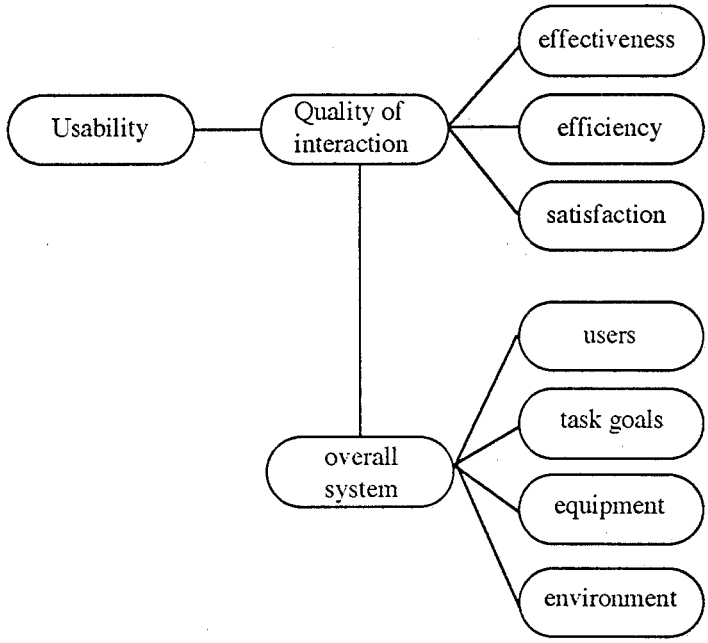

Figure 1.2: Usability components.

(2) the task goals that the system allows the user to achieve;

(3) the equipment, i.e. hardware, software and materials;

(4) environment, i.e.the physical and social environments which may influence the interaction.

System developers should perform different analyses on users:

(a) user characterization: it involves attempting to capture all of the information about the target user groups that is relevant to the proposed system;

(b) task analysis: it involves attempting to understand user's goals and activities, as well as tools they use and environment they work in;

(c) situation analysis: it involves an appreciation of the situations that commonly arise as part of the user's normal work activities, and a consideration on how these might affect both the user's needs and preferences.

(d) acceptance criteria: it refers to understanding users' requirements and preferences, which then form the acceptance criteria for the system.

The above mentioned analysis are related to psychology, ergonomics, artificial intelligence, ..., but, in order to be fruitfully exploited by designers, such analysis should express concepts in a clear and unambiguous way.

1.6 Formal methods in HCI

In HCI, all the contributions from different research areas need to be properly understood by all the partners of the design process in order to be correctly used. The use of formal methods, based on precise notations and mathematical models, allows the different partners which collaborate in the design process to provide their contributions in a precise, clear and understandable way.

Formal models may be used to describe the system, the user, the task.

Different formal methods have been proposed in the literature for modelling the system's interface behaviour [29, 1, 32, 15, 40, 68, 67].

Cognitive models [17, 13, 79] may be used to describe in a formal way users and tasks.

Architectural models [23, 71] are employed describe the structure of a usable system which can be refined in an executable version, still mantaining the usability propeties.

Focusing on the first of these three types of model, interactive systems may be formalised at three different levels [25]. They are

- dialogue specification and analysis;
- specification of individual interactive systems;
- generic models of interactive systems.

Formal methods may point out problems and inconsistencies of a system before it has been implemented. If all, or the main, problems are highlighted at the beginning of the development, then the following steps of the life cycle of an application software will need very few modifications and so the testing time will be reduced. The employment of formal methods does not guarantee the usability of a system. However, some formal properties (such as predictability, observability, reachability, ...) are necessary for usability, and if a system does not satisfy such principles, then it may be unusable in many circumstances.

Dialogue specification and analysis

In HCI, dialogue refers to the structure of communication between a human and a computer. It is extremely important to formalise and analyse dialogues in order to eventually find usability problems before the system has been implemented. There are different notations which may be employed to specify a dialogue. Basically, such notations may fall in two classes, diagrammatic and textual.

The diagrammatic notation is easy to read since the designer can see at a glance the structure of the dialogue. State Transition Networks (STN) [38] is one of the most widely used models. It is based on transition diagrams, which are diagrams made by circles (to represent states) and arcs (to represent transitions). Each arc is labelled with the name of an action. This notation is useful in order to represent a sequential and iterative portion of the dialogue; it is not the most suitable to represent concurrent situations; in this case, Petri Nets may be a more suitable notation [9, 10].
Textual notations are easier for a formal analysis; the most widely used are grammars and production rules.

Analysing a formally described dialogue, interesting properties may be verified. For example, with a diagrammatic notation it is easy to verify if the dialogue is complete or not; in fact, in the last case, there is an action which cannot be performed in some state. Another property that is easy to verify is non-determinism. If, starting from the same state, there is more that one arc with the same label, then the dialogue is non-deterministic for that action. Both completeness and non-determinism are properties related to the action. An interesting property related to the state is reachability. A dialogue has the reachability property if any state may be reached, that is, if the diagram which represents the dialogue is fully connected. Often, the consistency property is analysed for the dialogue described with a textual notation. In [79] the authors give an example on how to analyse consistency by applying Task-Action Grammar. A deep discussion on dialogue is in [32].

Specification of individual interactive systems

An abstract mathematical description of an interactive system, before its development, is useful to explore the validity of design choices. By reasoning on the formal description, rather than on the real system, the eventual problems are discovered at the beginning of the developing phase.

In [75] Christopher Rouff points out the seven issues which make a specification formal:

- written
- communicable
- mathematical
- precise
- unambiguous
- supports analysis
- supports reasoning and prediction.

He also provides three main reasons for using formal notations:

- they give a precise and unambiguous description of the functionalities considered;
- they constrain designers to clarify aspects of their design;

- they allow designers to reason on their systems.

Formal specifications can also be used for analysis and verification of interactive systems.

A system specification may be done using some existing notation, as Z or the Act-One languages. A specification may also be done only for some component of the interface, not for all the system.
Sometimes, an existing and general pourpose formal notation may not be suitable to specify components of the user interface. To this aim, some new notations have been introduced. Usually, they arise as a modification of an existing one, or by a combination of different notations. An example of these is given by ICO [10], which is a formalism based on Petri nets and on the object-oriented approach.
In order to understand the behaviour of some component of the interface, it may be convenient to abstractly describe such a component as an object and then to describe with a formal notation the behaviour of such an object. An interesting example is provided by the interactor model [36].

Generic models of interactive systems

Abstract models are used to describe properties of a class of interactive systems, not of a specific one. An abstract model may be employed during the first phase of the development of an application software, i.e. the users requirements. Usually, such requirements are informal, and it is difficult to map something informal in logical and realizable steps which have to be implemented. Moreover, if the system description is informal, we cannot say precisely if the system satisfies the requirements or not. An abstract model may help in filling the distance between informal user requirements and formal specification, distance which is usually called the *formality gap* [29]. An example of abstract models is the PIE model [27] which we will discuss in the following Section; other abstract models are given in [43, 24]

1.7 The PIE model

The PIE model is a black-box model. With a black-box model it is possible to describe the behavior of an interactive system only in terms of the perceivable effects, without taking into account implementation aspects. The PIE describes a system in terms of its inputs and of the corresponding effects. The input is a command c or a sequence of commands, belonging to a set

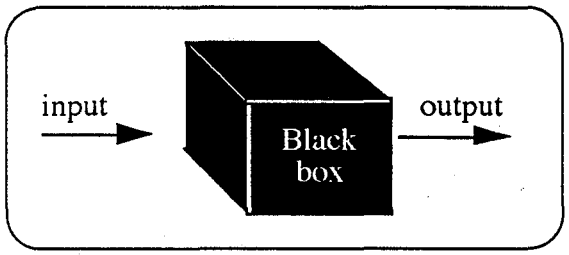

Figure 1.3: The black box model

of legal commands C. A program p is a command sequence, while P is the set of programs. All inputs are interpreted by a function I (Interpretation function), producing an effect e, which is the output; the set of the effects is E. Due to last three used capital letters, this model is called PIE. An important feature of this model is the flexibility of the space of the effects, which may be seen as the display observed by the user, or as all the content of a file, or as all the information available to the user. Also the set of programs is flexible. It may be seen as keystrokes or mouse movements or a sequence of more abstract operations of a specific domain provided by the user or the system. The choice of a black-box model has different advantages. The main advantage is that it represents an interactive system from the user point of view, that is without taking into account the internal structure. Moreover, it allows to express properties of interaction which are independent from the domain and from the implementation. Its behaviour may be mathematically represented, so the properties are precise and may be proved.

We can define a PIE as a triple $< P, I, E >$, or we can expand the interpretation function as follows:

$I : P \rightarrow E$

Each PIE may also be described in term of states and transition functions. Indicating with S the set of states, we can define the transition function *doit* as:

$doit : S \times C \rightarrow S$

starting from the initial state s_o. The *doit* function is a very simple way to describe interaction.

In order to obtain an effect from a state, we need a function, a projection from all the state information to that regarding the effect: $proj : S \rightarrow E$

In this manner, we can handle a PIE as a sextuple $< C, S, E, s_o, doit, proj >$ instead of a triple.

1.7.1 Predictability

One principle that an interactive system should guarantee is *predictability*, for which the user, starting from the current effect, should be able to predict the behaviour of the system once a commands sequence has been entered. At this point we have to introduce some definitions and properties regarding the commands sequences which are equivalent or seem to be so ([29, 32, 84]).

Given two different commands sequences p and q, we say that p is equivalent to q if they have the same interpretation. We can define an equivalence relation $\equiv_I$ for any PIE:

$p \equiv_I q \ \hat{=} \ I(p) = I(q)$

Most current systems allow the user to obtain a particular effect in more that one way, but, following more than one path, different internal states may be reached. When we are in this situation, we say that such an effect is ambiguous. Formally, indicating the concatenation of command with '$\frown$', we have:

$ambiguous(e) \ \hat{=}$
$\exists \, p, q, r \in P \ \ s.t. \ \ I(p) = e = I(q) \ \ and \ \ I(p \frown r) \neq I(q \frown r)$

The ambiguous effects are due to the different command histories.
If no effect is ambiguous, we say that the PIE is monotone.

$monotone : \forall p, q, r \in P, I(p) = I(q) \Rightarrow I(p \frown r) = I(q \frown r)$

The monotone property says that it is possible to predict the future behaviour of the system from the current effect. The last property allows us to introduce another definition of equivalence, the monotone one $\equiv^{\dagger}$. Given two different sequences of commands p and q, we say that they are monotone equivalent if they have the same interpretation and, once entered any command sequence r, they continue to have the same interpretation. Formally,

$p \equiv^{\dagger} q \ \hat{=} \ I(p) = I(q) \ \ and \ \ \forall r \in P, I(p \frown r) = I(q \frown r)$

With the monotone equivalence, what *looks* the same *is* the same. If the PIE is monotone, then we can talk about effects or states indifferently. In this case we have that $E \equiv (P/\equiv^{\dagger}) \equiv S$, where $P/\equiv^{\dagger}$ is the quotient of P on the monotone equivalence and S is the set of the reachable states. From the definition of monotone equivalence we have that

$p \equiv^{\dagger} q \ \ \Rightarrow \ \ I(p) = I(q)$

which means that $\equiv^{\dagger} \Rightarrow \equiv_I$. We may call $\equiv_I$ weak equivalence.

If the PIE is monotone, we can express the *doit* in terms of the interpretation function, as follows:

$I(null) = s_0$;
$\forall\, p, q \in P,\ doit(I(p), q) = I(p \frown q)$

1.7.2 Reachability

An important property of the PIE model is *completeness*. A PIE is complete if any effect is produced by the interpretation of a suitable program: this implies that the interpretation function is surjective. The completeness of a PIE represents the simplest *reachability* condition. With the term reachability we intend the possibility for the user to obtain a wanted effect starting the computation at any state. If, after a program p, the computation has been interpreted with an effect $e = I(p)$, and the user wants to obtain an effect e', he must find a suitable program r such that $I(p \frown r) = e'$. Formally, we say that

$\forall e \in E : \forall p \in P, \exists\, r \in P : I(p \frown r) = e'$

This implies that a PIE has the reachability property if the interpretation function I is not only surjective, but surjective $\forall p \in P$. The above definition is called *strong reachability*.

A special case of reachability is undo, with which the user can reach the previous state. If we consider P as a semigroup, provided with right program concatenation as the associative operation, the monotone equivalence is a right congruence. It is natural to study the full congruence, where the equivalence is preserved in all contexts. This equivalence is the *strong equivalence*, which we indicate with $\sim$. Two programs p and q are strongly equivalent if they produce the same effect, with the same history and with the same future action. Formally we have:
$p \sim q \ \hat{=}\ \forall\, r, s \in P,\ I(r \frown p \frown s) = I(r \frown q \frown s)$

With the canonical projection we move from P to the quotient $P/\sim$ for which the null command is the identity element.

Later we will suggest that undo is also a form of reachability. There are different kinds of undo and the strong equivalence will be very important in describing the required properties. Undo can be seen as a function which modifies the command history. One type of undo considered in [29] is the existence of a function

$Undo : P \to P$, such that $p \frown Undo \sim null$
such that with such an undo the quotient set $P/\sim$ is a group.

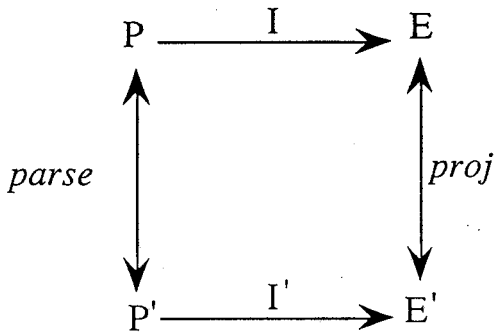

Figure 1.4: Isomorphism between PIEs.

1.7.3 Relations between PIEs

Given two PIEs, $< P, I, E >$ and $< P', I', E' >$, we say that an isomorphism between them exists if there exists two one-to-one relations, $parse : P \rightarrow P'$ and $proj : E \rightarrow E'$, such the diagram of Figure 1.4 commutes. A special case of isomorphism between PIEs is given by $< P, I, E >$ and $< P, I/\equiv_I , E/\equiv_I >$, in which *parse* is the identity and *proj* is a one-to-one function. Usually, the *proj* and *parse* are general relations, sometimes with restriction on *parse*. A special kind of relation is given considering both *proj* and *parse* as functions; the major cases follows:

- $1 - morphisms$: both *parse* and *proj* go in the same direction;
- $2 - morphisms$: *parse* and *proj* go in opposite directions;
- $0 - morphisms$: either *parse* or *proj* is one-to-one.

The $0 - morphism$ is a special case of morphism in which either *parse* or *proj* is one-to-one, that is 1-morphism and 2-morphism coincide. In Chapter 7 we will consider 0-morphism where *parse* is the identity. As this is the only kind of 0- morphism used in the thesis, we will use 0-morphism to refer to this case from now on.

An exhaustive discussion on PIEs and relations between them is given in[29, 28].

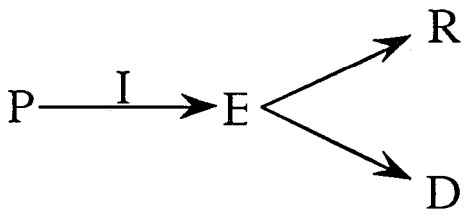

Figure 1.5: The Red-PIE

1.7.4 The Red-PIE model

The PIE model may be refined in order to distinguish between two different characteristics of the space of the effects: the result and the display. The result is the final product of the interaction between a human and a computer, while the display represents intermediate and ephimeral aspects of the effects.

The PIE refined with result and display is called $red-PIE$. This model is an enrichment of the original PIE with other two functions, $result : E \rightarrow R$ and $display : E \rightarrow D$, where R is the set of results and D is the set of displays. A diagram of this model is provided in Figure 1.5.

The PIE model has been used to analyse different properties of interactive systems. One of these is the WYSIWYG (what you see is what you get) principle, for which what the user can see is really what he gets. In the Red-PIE model, WYS is given by the display, while WYG is given by the result. Important properties of interaction may be characterised as relationships between result and display. One of these is the *predict* function,
$predict : D \rightarrow R$
$\forall\, s \in E$, $predict(display(s)) = result(s)$
for which starting form any display, it is possible to predict which will be the result.

The basic concepts of the PIE model, introduced in this Section, will be used in Chapter 4, 6 and 7 in order to describe properties of interactive systems with and without undo and the relationships between them.

Chapter 2

Interactive Systems

In this chapter, an interactive system is defined as a subset of a reactive one. HCI is a particular kind of interactive system, in which one of the involved entities is a human.

In order to allow a dialogue between entities which use very diverse languages (as it happens for humans and computers), it is necessary to support the communication with suitable *interaction languages*. Such languages allow the user to communicate with the computer in a way that is as easy and comfortable as possible, yet necessarily depending on the nature of the application, the user needs, skills, background, etc.

Differences between interactive languages and traditional ones will be highlighted. In particular, since the main feature of interactive systems is represented by the *feedback*, the explicit answer to any user action, emphasis will be given to the *pragmatics* in interaction languages accounting for the *interpretation of the feedback*.

Since visual interfaces may be seen as interaction languages which allow the user to reach high level of interaction, their properties are discussed in this chapter and a model of human-computer interaction based on the visual approach is proposed.

Finally, a table including the main characteristics of interactive systems is shown.

2.1 Reactive systems in computing

Sometimes the words *interactive system* have been improperly used, or as buzzwords, not always based on a full understanding of their meaning.

Most computer users think that an interactive system is a computer, provided with a visual interface and a mouse. Probably, this is due to the fact that visual interfaces and mice represent the main components of the most common and spreaded interactive systems. But it is not a mouse or a visual interface that makes a system an interactive one. To clarify what an interactive system is, it may be useful firstly to introduce the concept of *reactive system.*

The notion of reactive system is extremely natural, and, as well as all the innate and obvious things, difficult to express. It is a very general concept and may also be applied to different fields. In fact, many examples of reactive systems are offered by nature, particularly in the world of biology and chemistry. A broad definition considers a reactive system as a complex whole, a set of entities in which any component is able to provide a response to an external stimulus [70].

An *interactive system* is a special case of a reactive one: it may be seen as a set of entities which are able to provide a reaction to an external stimulus not only once, but many times, so generating a sequence of stimuli and responses between at least two entities.

In the computing world, an example of reactive system RS is given by the computer and all the entities, internal and external to it, which are able to give, and/or react to, external stimuli, external in the sense that they are provided by another entity of the same system. Also in this case, as in the real world, an interactive system IS is a subset of the reactive system RS [69]. Any entity of this reactive system may be seen as a process, an object, or what Milner [59] defines as an agent.

Examples of elements belonging to RS are those processes which are activated by system fuctions. In this case there is no dialogue, but simply a communication of a stimulus from a sender process to a receiver one, which is then activated.

Interactive processes are those processes that react one to the stimuli of the other and a continuous exchange of information, i.e. stimuli, between the two communicating processes, is present.

HCI is a special kind of interactive system, in which the two involved entities in the communication are a human and a computer. This means that the interactive system involved in HCI is given by *the computer* and *the user*, not by the mouse and the visual interface. The mouse and the visual interface represent tools, (which are not unique) for communication between the two involved entities.

Figure 2.1 shows the hierarchy of a reactive system in computing. The

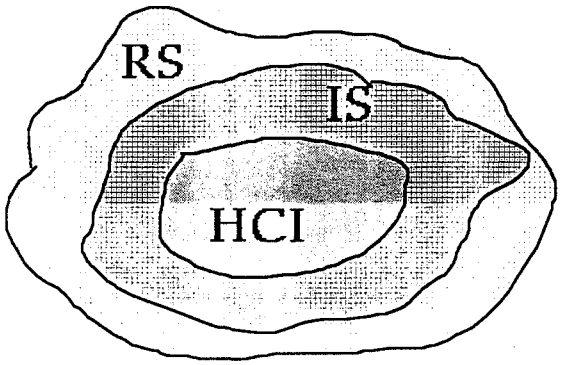

Figure 2.1: The hierarchy of reactive systems in computing.

external set represents the reactive system RS; the interactive system IS is a subset of RS and the inner set is HCI: in this way we have the following hierarchy: $HCI \subseteq IS \subseteq RS$.

Since the aim of this thesis is to focus on aspects related to HCI, in Figure 2.1 we may also skip the interactive processes that do not involve humans (for example agents in expert systems). In this way the hierarchy is reduced to $HCI \subseteq RS$ (see Figure 2.2).

In the following Sections, dialogue [32] and communication aspects in HCI will be discussed.

2.2 Dialogue

Each entity involved in any communication requires the ability for reciprocal comprehension in order to partecitate to a dialogue. Such a dialogue happens in a natural way when the involved partners are humans that use the same spoken language. This becomes more difficult, or even impossible, when the communication occurs between entities that are using different languages: the more "distant" the languages, the more difficult the communication.

Within HCI dialogue refers to the structure of the communication between a human and a computer. The kind of communication to which most humans are used is the verbal one, employing a spoken language; on the con-

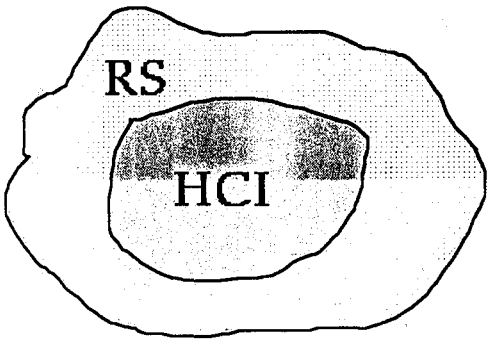

Figure 2.2: HCI is a subset of reactive systems in computing.

trary, the language of the computer is made by 0 and 1 strings. Since humans and computers use two different languages, in order to allow communication between them, it is necessary to provide a support to the interaction: a suitable interface.

The interface is the medium [33] through which such a communication takes place. There is not a unique interface style, and, the reason for the existence of such a plurality is due to the fact that interfaces change with evolving technology. During this transformation, slower at the beginning and faster in recent times (because of the speed of technology changes), interfaces gradually evolved, passing through the command-line, question-answers, form-filling, menu interfaces [80, 44], until the actual visual approach and virtual reality environment present today. The interfaces evolution does not imply that the past interfaces may no longer be employed. In fact, the choice of the most suitable interface depends on the aim of the application, on the kind of task that the user has to perform, on the users class, etc. The change of interface styles is the result an attempt to improve HCI, making it as close as possible to the human-human communication.

Any interface style is an *interaction language* [85]. The more natural and easy the language, the higher is the interaction level that a human can reach while interacting with a computer. By level of interaction we mean a kind of dialogue ordering: the closer human-computer communication is to human-human, the higher the interaction level. Traditional programming languages

and the last generation of interaction languages -the visual approach and virtual reality environment- represent the opposite poles along the rank of interaction levels, while all the other interface styles are in the middle.

Of course visual interfaces and virtual reality, the last generation of interaction styles, are not the final result of this research, but certainly in the future a further evolution of an interaction environment will be provided. Actually, one of the most common interface styles, combined with the use of a mouse, is the visual one, due to the fact that this approach makes interaction as simple as possible, basing the communication on the essence of interaction, on the action-reaction cycle. In the action-reaction cycle, the user sees an object, provides his input and observes the result produced by the computer, so becoming the integral part of a loop [31].

Moreover, interactive systems are not a barrier for handicapped people. On the contrary, interactive systems, provided with suitable devices, may represent an important step to allow handicapped people to become more and more part of the society. In fact, depending on the nature of the handicap, different interaction languages may be provided. Particularly, systems suitable for blind people are based on audio tools [74].

2.3 Interaction languages versus traditional ones

A comparison between a spoken language, an interaction language and a traditional programming language may well show the power of interaction languages.

Any spoken language is based on words as a sequence of elements belonging to a finite alphabet of symbols. Such words may represent any of the nine parts of speech: noun, adjective, article, verb, etc. In traditional programming languages we have elements with a role similar to the parts of speech: a variable as a noun, a pointer as a pronoun, a data type as an adjective, a function (in a broad sense, considering both subprograms and operations) as a verb, etc [37].

Using a spoken language, we do not declare any part of the speech; for example, before using the word "tree" in a conversation, we do not say that "tree" is a name. We simply use that word, without any declaration, because the fact that it is a name is implicit, it is given by the (previous) knowledge of the grammar of the employed language.

When using a programming language, the compiler or interpreter recognises the lexicon and syntax and checks the correctness of the lexical and

synctactical forms, but the computer does not know whether $p1$ is a pointer or a function, if $p1$ has not been declared in the program.

When using whatever interaction language, as in a spoken language, there is no declaration of any "part of the speech" by the user. This means that there is no explicit knowledge of the lexicon, because the last is naturally provided by the interface, as fields the user has to fill, or menu items, or icons, etc. depending on the kind of interface style.

Less immediate may be how to properly handle these elements, that is the syntax. Employing a traditional programming language, any user has explicitely to learn, by consulting books or manuals, the syntax of the chosen language, employing time and, often, much effort is required. Moreover, a correct syntax may not ensure that a prefixed goal may be reached: this may only be done with a correct interpretation of the statements, that is the semantics. So, before starting to program, a user has to learn the lexicon, syntax and semantics of the chosen programming language.

Conversely, while using an interaction language, the syntax is driven by semantics, in the sense that the order of execution depends on what the user wants to obtain. In this case, the interaction itself is task-driven, because any user action is motivated by what the user is interested in achieving. In particular, for interfaces styles that allow a high level of interaction, the syntax is immediate, and, in only a few situations, the system forces the user to do some actions following a well established execution order.

2.4 Pragmatics in Interaction Languages

When using a traditional programming language, the pragmatics, i.e. the interpretation of computation results and their consequences, has not been considered strictly related to the lexicon, syntax and semantics of the employed language. We generally talk about syntax and/or semantics of instructions, while we talk about results of computations. Due to the nature of traditional programming, it is not possible to use partial results in order to provide the next input; in fact, the computation result is interpreted at the end of the computation itself and the programmer may agree with it or not; if not, some changes may be done on the data and/or instructions before starting again the computation. Pragmatics is also implied in this case -with the interpretation of the results- but is considered external to the computation itself, in the sense that the programmer can use it only at the end, not while the program is running.

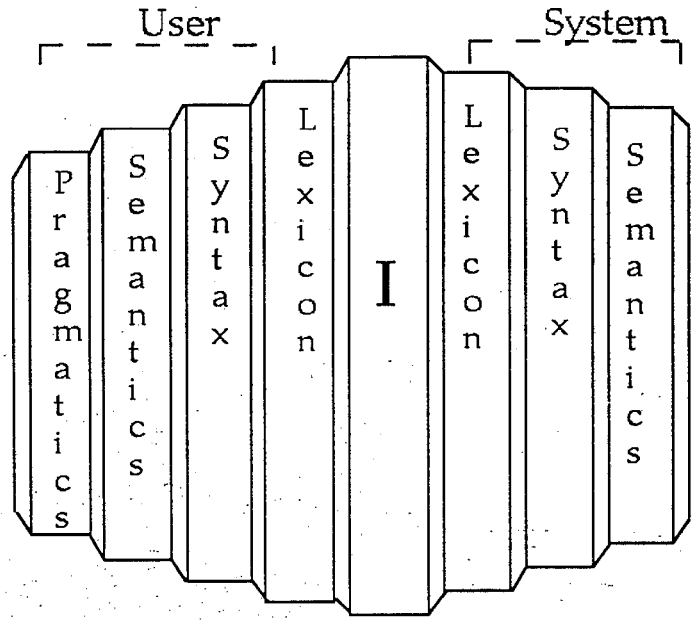

Figure 2.3: Different dimensions of the interaction language and system language.

On the contrary, within interactive systems, the interpretation of results plays a very important role, since it is the "engine" of the interaction.

Actually, in interactive systems it is not only important to have a correct interpretation of the user actions, but also of their consequences. Such consequences are not merely the feedback, but *the interpretation of the feedback*, i.e. the pragmatics, depending on the physical working environment, the user aim, knowledge, expectations, etc. The result of the interpretation is the basis for the successive user action.

In Figure 2.3 the dimensions of interaction and system languages are schematised. The same user action has a double interpretation: one is from the system point of view, and generally humans ignore it, while the other is from the user point of view, which is exactly the one in which the user is interested. In the pragmatics, as the interpretation of the feedback, is intentionally lacking in the system language, so the schema results unbalanced. In fact, if we introduce pragmatics to the system's language, we may

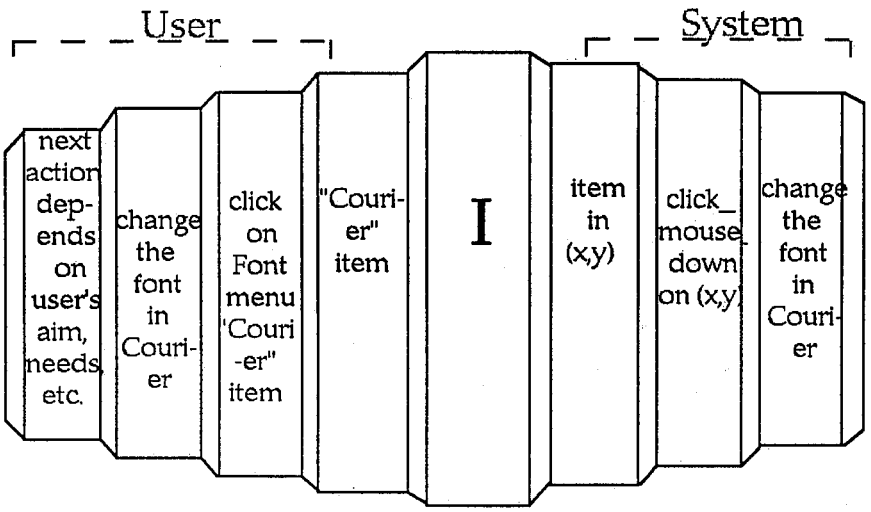

Figure 2.4: An example of the different dimensions of the user and system language.

imply that the system is adaptive [19], yet the purpose of this thesis is not specifically to deal with adaptive systems, but, more generally, to interactive ones.

By using interface styles which allow high interaction level, the system may be considered as a black box.

Employing a visual interface, for example, it is not important for the user to know that, working with a word processor, if he clicks on the menu in the location (78.45, 182.12) he can change the font of the selected text from the actual one to Courier, but it is important to know that if he wants to change the font of the selected text he can choose, from the Font menu, the font he needs, for example Courier (see Figure 2.4). This not only means that the interaction is semantic-driven, but also that for the user the system is in fact like a black box: he is only interested on what he can provide as input, and the system could return as output, on the surface of the black box, i.e. he does not care about the contents of the box itself [46].

2.5 The visual approach

As said in the previous Sections, visual languages, provided with a suitable device (very often a mouse) allow the user to reach a high level of interaction. In this Section, we are going to account for the power of visual languages.

In the field of computer science, the term 'visual' refers to something which is represented in a graphical, bidimensional manner, in contrast with a linear, textual one. For example, in a command line interface a text document is specified only by its name (textual, linear representation), while in an iconic interface it is indicated by an icon, which is graphical and bidimensional. Visual languages allow the user to directly handle instructions and data, both visually represented by the interface of the employed application software. One of the main features of the visual approach is that it is more immediate with respect to the traditional one: actually, with a simple glance, it is possible to know which objects we can handle and also their different nature (for example, icons representing files are different from the ones representing folders or applications).

Moreover, such an approach is based on the visual representation of both objects and language constructs, making it independent from the user native language and so being usable by a wider number of people.

The *lexical level* of a visual language may be considered as the set of visual signs which one user can perceive and understand by looking at the screen, and which may be provided by the user himself as input or by the system as output. Such visual signs (buttons, geometrical shapes, icons, menu items, etc.) are handled by the user simply as data or program functions, without any particular knowledge about them. Under the surface of the application software, i.e. the interface component communicating with the computer, the programmer makes a clear distinction among the diverse nature of the displayed objects, with an explicit declaration of data types, links, subroutines, etc.

The actions which one user can perform, which are the events, are very easy to produce: it is possible to click once or twice on visual objects, or to drag them, or to enter commands by using the keyboard.

The user does not need to learn the syntax of the visual language (eventually it is the lexicon which may create more trouble: in fact, the user may have some problem in understanding the metaphor that is associated to any visual object [18]), because it is immediate; he simply navigates through the system producing events using a suitable device. If the user himself tries to perform a forbidden action, as when if he is following an incorrect syntax,

the system will highlight this situation by opening a dialogue box, or by means of a "beep", or simply by not executing the last entered command.

When the syntax is not immediate and needs some other specification, the system usually refines the dialogue with a submenu or dialogue boxes.

We can say that the *syntactic level* in visual languages is represented by the legal arrangements into which visual signs, displayed on the screen, can be organized by the user as input or displayed by the system as output.

The sequence of user actions, that is the interaction history, is strictly related to the semantics of the actions themselves. In fact, any user event is motivated by what the user himself wants to obtain: for example, if he is interested in opening a file, he can select the item "Open" from the menu title "File", or may enter the equivalent command with the keyboard, or by a double click on an icon. Along this way, the interaction is driven by semantics, since any user command is chosen according to its interpretation from the user point of view. In a semi-formal way, we can say that the *semantic level* represents the meaning of the communication, the effect of the user actions. As a consequence, from the system point of view, such a level represents what *the system is going to do*, while from the user point of view it represents what *the system should do*, after having received an input.

In HCI the importance of pragmatics is directly proportioned to the feedback that the system provides to any user action. For this reason, the more the interface style allows a high level of interaction, the more the pragmatics become important, since the pragmatics provides the interpretation of the feedback, in order to establish the next user input. In visual languages, in a semi-formal way, we can say that the *pragmatic level* represents the perception of visual signs on the interface and their understanding by the user.

A model of human-computer interaction, based on the visual approach, is proposed in Figure 2.5. In this model, syntax, semantics and pragmatics are merged together in the user mind, while the visual representation of elements (lexicon) is displayed on the interface.

From the user point of view, when he is using a visual language, he is handling the data directly, without any other medium in between. Instead, he is performing a dialogue through a language and, what he imagines as data, is really a *visual representation of the data* on the surface of the application software with which he is working.

We can say that the power of the visual approach is in the reduction of the gap between the user language and the system language, allowing an interaction as easy as possible so increasing the usability of the application

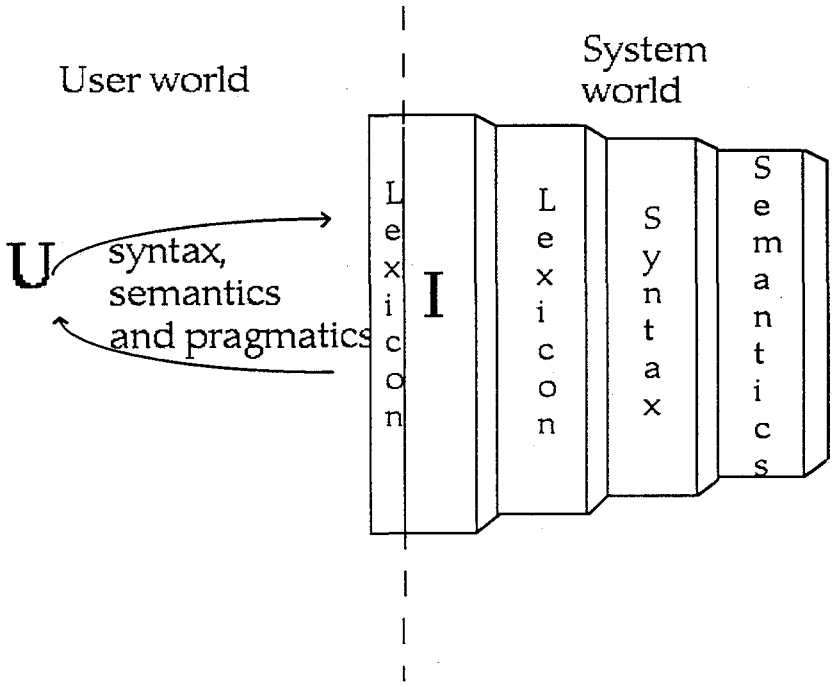

Figure 2.5: Interaction and system languages.

software [57].

2.6 Features of interactive systems

In the last Sections, we informally described the dialogue, the way in which a user can provide his input to an interactive system, but we have not said what an interactive system is. In next Chapter we will provide a definition, while in this Section we are going to describe an interactive system in terms of its main feaures, which differentiate it with respect to a traditional system. To do this, we consider both the perceivable features (as the feedback, recovery functions, ect.) which a user can perceive simply interacting, and the internal ones. The last are related to the code structure, so that only the programmer knows them, but their presence strongly influences and chracterizes interaction.

Looking at an interactive system from a programmer point of view, its code presents a strong modularization. This fact was and is present also in traditional programming languages, but only as a functional structure, without any direct consequence for the user. In fact, such a modularization is at the processes level: its effects are only inside the black box.

Conversely, in interactive systems, the presence of modularization assumes a different importance: it is used here to break down a problem into many subproblems, as much as possible independent each one from the others. Such subproblems are small blocks of code, which may represent agents or objects and, depending on the adopted interface style, are represented on the surface as fill-in forms, menu items, icons, buttons, etc. Differently from traditional programming languages, with interactive systems the user can exploit the effect of modularization. In fact, the code modularization happens under the surface of the application software and the user does not realise what is happening. But, after the interactive application is started, the user can provide his input to the computer, employing event-driven and/or an agent-based approach, by choosing the order of the modules execution, order which depends on what he needs/wants to do. This means that now the computation is not forced anymore to follow the way envisaged by the programmer, but is *driven by the user*. Naturally, the user cannot always perform everything in the order he wants, since he has to do any actions following the right syntax. But the constraints that have to be respected are usually more or less a "natural" (depending on the interaction level) representation on the computer, step by step, of the task the user is going to

perform. We can say that the main consequence of modularization in interactive systems is that, by being present on the interface, it may be exploited by the user in order to drive the interaction.

Synchronization, as well as modularization, was and is present in traditional systems too. In traditional systems, synchronization problems are related to system processes and those generated by the execution of a program. However, the processes generation is pre-planned and usually a good utilization of semaphors and monitors may be enough to guarantee a correct system functioning by handling critical sections. In interactive systems the situation is similar, except for the fact that processes may be also activated by the user, without any pre-planned strategy. This means that the number of concurrent processes and synchronization problems may be higher in interactive systems.

The above mentioned properties are not enough to describe interactive systems, not only because the idea of modularization is old and imported by the traditional programming approach, but also because, in this way, many systems, such as the Unix operating system, may be classified as interactive, or highly interactive, because of its inherent modularization.

In fact, the main consequence of modularization is the feedback, the fact that the user can have an immediate knowledge about the computer answer to his previous input. Such feedback is the computer answer to the execution of a module and it is not the final result of the whole computation, but instead any feedback represents a partial result.

For this reason, operating systems without an immediate communicated reaction to any user action may be considered simply as reactive systems, not interactive ones. But, since by employing some suitable commands, the user can have some information on the system state, he can have a kind of indirect feedback, for this reason such operating systems may also be considered interactive, but with a very low level of interaction (as Unix).

The role of interactive systems changed with respect to the employment of traditional computers. In fact they are not used anymore for numerical computation only, but particularly as a tool which helps users in their daily work; which allows communications and cooperation between users (also far located); which may be employed as a navigational tool to explore the system.

Comparing the traditional way of computing and the one performed with interactive systems, in the last we no more have an initial and a final state of the computation but, if we consider the initial state as the opening of an application or file, and the final state as its closing, in interactive systems

we also have a lot of intermediate states: the partial results of the computation. Moreover, the interpretation of the partial results (pragmatics) is very important during the interaction because, depending on those, the user will decide which will be his next action.

Furthermore, it is also important for the user to obtain a computer answer to his input in an acceptable amount of time. Also for traditional programming, timing aspects are important in order to reach the end of a computation (the result) as soon as possible. But in interactive systems a delay in answering not only can annoy the user, with a consequent change of his attitude with respect to the computer, also influencing the usability of the application with which he is working, yet it can also cause a breakdown in interaction [30].

As mentioned before, the aim of interactive systems is different with respect to traditional ones, and special attention is given to the fact that they allow the user to navigate through the system. Employing the traditional approach, not only there is only one user, the programmer, but also his activity is totally pre-planned, since every forseen action is represented within the code and the user can not modify the program execution. The visual approach, instead, combined with direct manipulation, allows the user to have an exploratory attitude while interacting. In this way the user can have information on how the system is functioning and on what he can do when using such a system.

The ability of the user to navigate through the system allows him to have more information on the system functionalities, being so able to employ it correctly. When the user does not know how the system is functioning, he cannot do prediction on its behaviour: this means that the system is non-deterministic from the user point of view. Moreover, because interaction is driven by users and there is not a physical law which controls the human behaviour, the interaction is non-deterministic, even if the computation is still deterministic. Since there are two communicating partners in HCI, non-determinism is double: from the computer point of view, when it is not able to foresee human behaviour, and from the user point of view, when he is not able to foresee the system behaviour. Aspects of non-determinism will be discussed in Chapter 4.

Within an environment in which the user at any step decides the way to be followed, he may realize immediately whether the last performed action is wrong or not, that is whether the last reached state is the expected one or not. If not, in order to repair the error, the user needs a system function which allows him to delete the effect of the last performed action(s), so

reaching a past state.

The user can realise that the reached state is wrong if he knows how the system is functioning. If he does not know it, the user can then exploit the visual approach combined with direct manipulation in order to navigate through the system. In this case he cannot realise if the reached state is wrong or not, but he can decide if the reached state is useful or not to him. In case it isn't, the user needs to change the direction of interaction, going back in the interaction history and following, from that point, another way.

If the user thinks he knows how the system is functioning, but he reaches a state that is different from the one he was expecting, then there is a mismatch between the two semantics, one from the user and the other from the system point of view. In this case, a system function which allows the user to reach a past state, not only allows him to repair errors, but also increases his knowledge on the system behaviour.

At this point, the importance and the usefulness of a specific recovery function to be applied after any feedback is quite clear: the *undo* functions. Such undo is explicit when an *Undo* function is explicitely available on the interface of the application software with which the user is interacting, implicit when it is possible to obtain the same effect of an explicit undo, but by employing other interactive functions.

Last, but non least, there are usability aspects [5]. While in the Sixties and Seventies the usability of an application software was a desirable, but not necessary quality, since the Eighties usability becomes a *compulsory* feature. In order to ensure that a system is effective, efficient and easy to use [52], real users have been more and more involved in the system development.

Figure 2.6 collects all the above mentioned properties. Such properties are qualities, attributes, internal code and system structures, ... but there is only one system function which characterizes interactive system with respect to the traditional ones: the *Undo* function. Being linked to the feedback, undo may be present only in interactive systems, while in traditional ones we can have only recovery functions (see Chapter 3). Moreover, by allowing the user to have more information on the system behaviour (when the user navigates through the system) and/or on its internal structure (when there is a mismatch between the user and system point of view), undo allows the user not only to delete the effect of the last past action(s) but also to handle and resolve non-determinism during interaction. In the following Chapters, undo will be deeply discussed, initially as a system function from the user point of view, and, successively, discussing its formal properties.

systems / properties	Traditional	Interactive
modularization	(functional) desirable	(event-driven, agent-based) essential
synchronization	present (system processes)	present (system and user processes)
aim	computation	computation, daily work, communication, navigation tool
computatior states	only initial and final states	initial, intermediate and final states
feedback	no	yes
pragmatics	not really present	strongly present
timing aspects	important but not basic	basic
user behaviour	pre-planned	exploratory
system behaviour	deterministic	computation deterministic, interaction non-deterministic
recovery	only recovery functions	recovery functions plus implicit / explicit undo
usability	desirable	a compulsory property

Figure 2.6: A comparison between traditional and interactive systems along a set of significant properties.

Chapter 3

Recovery Functions

Undo is a special kind of recovery function. Since it is strictly related to the feedback that a computer provides to any user input, we can talk about undo only in interactive systems.

A differentiation between reactive and interactive systems is proposed and, for both, recovery functions are discussed. Such functions are ordinary recovery functions, implicit and explicit undo.

A lot of emphasis will be given to the automatic recovery functions in the databases environment, in order to introduce the definition of checkpoints, necessary in the following Chapters to discuss in depth the undo function and its internal structure.

Finally, in the last Section, undo in collaborative work will be introduced in order to provide a description of undo also in a multi-user environment.

3.1 Different kinds of recovery functions

Much work has been done on the undo, but until now it is not very clear how we have to deal with it and which way has to be followed when developing an interactive system. Really, undo represented a big problem since the beginnig of the use of computer [45], also for systems of the 'old generation', used particularly for computational problems, and very far from the idea of interaction which actually is natural while using a computer.

But what is the undo? Is undo a function, a command or whatever? In [3] Abowd and Dix make a clear distinction of the meaning of undo from the user and system point of view. From the system point of view, undo is a function, and as one, does something; from the user point of view, undo is an

intention, the intention to recover a past situation. This recovering could be done employing any system function, not only undo. Nevertheless, it would be useful for the user to have a suitable tool helping him in pursuing his aim. This tool, as we will see in the following, is the explicit undo function.

An important feature of RS in computing is that they provide functions (manual and/or automatic) which allow to repair errors whatever they occour. These functions are usually called *recovery functions*, because they are used to recover a past state.

The idea to have a computer provided with a tool which allows to change the past actions is not new. Turing himself in 1937 described an hypotetical machine able to perfom any symbolic computation that a mathematician can do with paper, pencil and rubber (from [84], pag. 84). Actually, we have different kind of recovery functions, depending on how, when and on which objects each one of them acts. Any function which changes the system state from the actual to a previous one, is a *recovery function*. If the user can change the state by himself while interacting, we are in the presence of *undo functions*. This undo is *explicit* (see Section 3.4) when it is available on the interface as a menu item or as a button, it is *implicit* (see Section 3.3) when it is possible to reach a past state by using some interactive system functions, but not an explicit undo.

In HCI we can distinguish two kinds of interaction: interaction with the operating system and interaction with an application software. When a user is interacting with an operating system, he is acting at the file level, i.e. the objects he can handle are files. When he is interacting with an application software, the object of interest is the content of a file.

In RS only ordinary recovery functions are available. At the file level, ordinary recovery functions and implicit undo are available. Within an application software, both implicit and explicit undo are available. Moreover, within an application software it is also possible to perform any ordinary recovery function since, by applying some system functions, the working environment may be changed, moving from the application level to the file one, in which any ordinary recovery function may be used.

The above considerations allow us to refine the hierachy of interactive systems proposed in the previous Chapter by differentiating HCI in HOSI (Human-Operating System Interaction) when interacting with an operating system, and HAI (Human-Application Interaction), when interacting with an application software. The resulting hierarchy, based on the availability of recovery functions, is shown in Figure 3.1.

Considering the different kinds of recovery functions and the just pro-

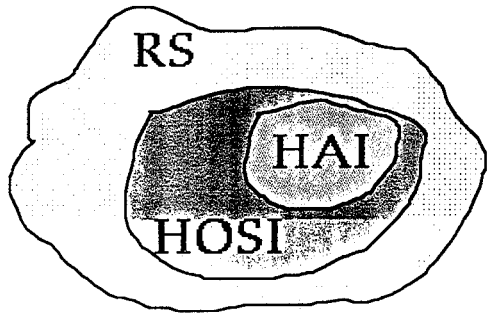

Figure 3.1: The hierarchy of reactive systems in computing.

posed hierarchy, the following proportion holds:

Recovery Functions : $RS = Implicit\ Undo : HOSI = Undo : HAI$

In the following Sections, recovery functions, implicit and explicit undo will be discussed.

3.2 Ordinary recovery functions

When we talk about the possibility to reach a past state in a reactive system, we mean the possibility to handle data that have been already used. In order to do this, it is possible to employ two kinds of recovery function: one is manual and the other is automatic.

Manual recovery functions are system's functions activated by humans; even if there is a human, there is communication, not interaction, between him and the system. This is due to the amount of time which elapses between user-action/computer-reaction and the following user action. A typical manual recovery function is given by the functions *backup* and *restore*, used respectively to back up data on secondary storage, and to restore them. The *backup* function copies the file(s), provided as input to the function itself, from the disk to a secondary storage device (generally floppy disk or magnetic tape). If, while working, the user realises that his data are not correct, for problems due to the computer or simply because he realises that there is a mistake in what he is actually doing, he can re-use the old data saved by the backup. To do this he can employ the *restore* function, that

copies the data from the secondary storage, used for the backup, to the disk. The time which elapses between the two actions *backup/restore* is usually long (weeks, months or years!); for this reason we have symply reaction, not interaction between a user and an operating system when employing such recovery functions.

Mechanisms for automatic recovery are system programs that check the system status : if they found inconsistencies, they try to resolve them, changing the actual system state into the last right one.

An example of these functions is represented by the field of database systems [11]. Broadly speaking, we can say that the scope of the databases is the data storage and retrieval, mantaining data consistency. If a datum is added to the database -removed or updated- in order to mantain data consistency, the same modification has to be done at any occurrence of the same datum in the database. This means that a logical operation may be composed by more than one instruction. The collection of instructions necessary to perform a single logical operation is called *transaction* [82]. Actually, we can see a transaction as a sequence of read-write operations, plus a "label" (abort or commit) to indicate the result of the transaction. The meaning of a *committed* transaction is that the transaction terminated successefully, while the meaning of an *aborted* transaction is that some logical error occourred during the transaction. Since the data resides in the stable storage, while any modification is done in the volatile storage, it is important to save in the stable storage all the modifications caused by the transaction. To this aim, periodically, there is a *checkpoint*, that is the new value of the data will be saved in the stable storage.

If a transaction aborts, or a system failure occurs, some data may have been updated and some other not, because of the incorrect execution of the transaction: for this reason it is necessary to restore the data to the value that it had before the transaction started. In this case we say that the transaction has been *rolled back*. In order to restore a previous situation, it is necessary to mantain information regarding all the modifications done during the execution of a transaction. These informations are collected, as a sequence of records, in a data structure called *log*. Before a transaction T_i starts its execution, the record $< T_i \ starts >$ is added to the log. Then, for any *write* operation, a record is added to the log, mantaining information on the name of the transaction, the name of the data item, the old and new values of the data. The write is executed after the correspondent record is added to the log. After any checkpoint, a $<$ *checkpoint* $>$ record is added to the log. If the transaction T_i terminates successefully, a record

$< T_i$ *commits* $>$ is added to the log.

If the system has a crash, we may be in one of the following three possible cases:

1. the record $< T_i$ *commits* $>$ is present in the log before the record $<$ *checkpoint* $>$, this means that all the updated have been recorded in the stable storage and there is no need to redo, that is to perform the transaction again;

2. the records $< T_i$ *starts* $>$ and $< T_i$ *commits* $>$ are both present in the log after the last checkpoint, so the effect of the transaction has not been recorded in the stable storage and the transaction has to be executed again, that is the system invokes the redo procedure;

3. the record $< T_i$ *starts* $>$, but not $< T_i$ *commits* $>$, is present in the log before the checkpoint, then some data have been updated in the stable storage and some not; so, in order to maintain consistency, the undo procedure is invoked to recover the past value of the data.

The *undo* procedure removes the effect of the last transaction if not all the data have been updated in the stable storage. Naturally, the implementation of the undo depends on the kind of the application software, but the meaning of the word "undo" is the same as in the interactive application, i.e. to cancel the effect of a performed action. In this case, to delete the effects of action(s) performed after the checkpoint.

The *redo* procedure executes again a transaction whose effects have not been recorded in the stable storage at all. There is no strict connection between undo and redo, in the sense that redo is not the inverse of undo, but there is a loose connection among them, since undo and redo are related both to transactions "not committed" and "not stored'.

The procedures "undo" and "redo" above mentioned are *recovery* functions, not *undo* functions, as it will be explained later in the following Sections. In this case they are automatic functions; it is the system that decides when and whether to apply them.

3.3 Implicit undo

When a user is interacting with an operating system, as for example the Macintosh one, he has an immediate feedback (indirect as in the case of the Unix or direct and immediate as in the Macintosh environment) for any of

his actions. Because of the feedback, the user can immediately realise if the reached state is wrong or not. In case the reached state is wrong, he can use some system functions to cancel the effect of a past action. This may be done implicitly, in the sense that no system function is explicitly available allowing the user to revert the effect of an action he has performed. For example, in the Macintosh environment, if the user has just moved a file from a directory A to a directory B dragging an icon between two different windows, and he wants to revert this action, he cannot use the explicit undo function because it is not available on the menu, but he has to drag the icon from the actual window into the old one, so he has to perform the inverse operation. Also for an *Open* (file) or *Close* (always file) function an explicit undo is not available, but in order to delete the effect of an *Open*, the user can perform a *Close* and vice versa. Confirmation boxes are also examples of implicit undo. A common example of confirmation is after an *Exit*, when a dialogue box *Exit now?* with 'Exit' and 'Cancel' as option is opened. Pressing the 'Cancel' key the user can implicitely undo his *Exit* action.

At the file level it is easy to delete the effect of a performed action using the implicit undo. Why implement an explicit undo, which could be very espensive in terms of implementation effort, memory and execution time required, when it is possible to obtain the same effect without any extra memory or programming effort, and, particularly, in a way which is immediate and easy to the user? This is the philosophy that is behind the implicit undo. However, there are functions for which an implict undo is not available. An example is given by functions which allow the user to have more information on a file or directory. Since the user is receiving information, this action cannot be deleted, because it is not possible to remove information from the user knowledge. The change should be done to the user's knowledge, not to the object which is still the same. The fact that functions which allow the user to have more information are not undoable in some sense is natural and useful. Conversely, it is not very usefull that "delete a file" has no undo, neither implicit nor explicit[1]. Most of the actual systems ask a confirmation before deleting a file (as the *Empty trash* function), but a confirmation is not enough, since a user could make a mistake and, non intentionally, remove the file. However, being $HSOI \subseteq RS$, the user can apply a recovery function (backup-restore), provided that he has a copy of his file .....

[1]In the Macintosh environment, the deleting of a file is done by moving the icon, representing the file, from his position into the trash (assuming that it is not yet in the trash!). This action has as implicit undo - moving the icon out of the trash. The *Empty trash* function, after confirmation, effectively deletes the content of the trash.

Some operating systems also offer another kind of implicit undo, that is the *Stop* function. Since the main feature of interaction is the immediate feedback, the user may sometimes realise that his last performed action is wrong before that the new state has been reached. A typical example of this is given by a double click on an icon producing the opening of a file or of an application. The *Stop* function, allows to stop the running process without waiting until the application is opened in order to close it (*Netscape* has such a function).

3.4 Explicit undo

Usually, when we are interacting with an application software, an undo function is available on the surface as a menu item or as a button. Such a function is explicit, since the user can identify it (by reading the name or by recognising the icon) and employ it when necessary.

The functions belonging to the set of *explicit undo* are undos, redos and mechanisms which allow the user to browse the past history. The plural (undos, redos, browse mechanisms) is used because we can have different implementations of undo functions, depending on the kind of the application software. Undo functions are usually classified into two groups, one for which undo is self-appliable (see Chapter 5), that is the effect of an undo may be cancelled applying a following undo (flip undo), and the other for which undo is self-appliable, that is undo of undo may be used as a backtrack tool (backtrack undo) [56].

Flip undo is employed in the most popular Macintosh and Windows applications: the user may cancel only the effect of the last performed action. If no redo function is available, a subsequent invocation of an undo, immediately after a previous one, has no effect. Nevertheless, if redo is available, it is possible to restore the state into which the system was before the invocation of the first undo of the eventual sequence.

Suppose, for example, that a user is working with a word processor and he is writing the sequence of characters A B C D E. The typing of any character may be considered as an action, so the action history is given by the same sequence. The invocation of the undo modifies the history in A B C D; if no redo function is available, then any other undo invocated at that state is idempotent. If a redo function is available, as an explicit redo or as the undo of undo, then the user can reach again the previous state modifying the history in A B C D E. This is simply an external behaviour,

by considering the single undo from the user's point of view. We really need to have more information on the system state. This situation, together with the backtrack undo, has been introduced in [56] and will be plentifully discussed in Chapter 5.

With the *backtrack-undo* we intend a kind of undo which allows the user to delete the effects of the past user's actions starting from the last performed one. The iteration of undo may be employed as a backtrack tool which allows the user to go back along the action history, until reaching eventually the initial state. The backtrack-undo is used in many popular application, such as $Netscape^{TM}$ and $MicrosoftWord^{TM}$6.0. In the situation of the above mentioned example, the successive undo invocations modify the history in this way: A B C D, A B C, etc. until, eventually, reaching the initial state.

Besides the explicit undo, any other system function that allows the user to reach a past state is implicit undo. Consider, for example, again a text editor, in which, after having typed a word, it may have the same effect to perform an explicit undo, or to select the word and perform a *Clear*, or to select the word and type *Delete*, or directly type *Delete* as many times as the number of characters to be deleted.

In HAI the undo function assumes a very important role, more than in RS or HOSI. In fact, its importance is due to the feedback. The more the feedback is immediate and evident, the more it is important for the user to have a tool which allows him to recover a past state. The fact that the user can employ the undo, allows him to navigate through the system in a comfortable and confident way, since he knows that, if he performs some mistakes, he can correct them. So undo seems to be a magic tool, which increases the power of interaction and the user thinks he can do everything, because, if he is wrong, he can use undo. In practice it is not so, and it often happens that when the user tries to perform undo, this function is not available at that time.

3.5 Undo in collaborative work

Undo functions, as we will see in the following Chapters, are very complex functions and, for this reason, it is better to analyse them starting from a single user environment, in which problems of concurrency and conflicts for the resources are reduced. In this thesis, undo in collaborative work will not be deeply discussed, but, in order to provide a wide description of undo in different contexts, we are going to shortly introduce it.

With the term *collaborative work* we intend at least two users that share data and cooperate through a computer. In this case the users may have problems of resources, if both of them require access the same data at the same time; problems due to delays, if the users are distant and one of them tries to update a datum which really has been already modified by another user; problems due to the undo. In fact, after employing undo, which one will be the deleted action? The last performed action, but performed by which user? Or the last performed action in a global sense? The answer to these questions is given by the kind of undo implementation, which may be global or local. With local undo, any user has an 'independent working life' and the invocation of undo removes only his last performed action, while with the global undo the last action performed in all the document activity will be undone, independently from which user performed it. The global undo seems to be easier to implement, since the system needs to keep trace only of one stream of user's actions, while for the local undo the system needs to keep trace of as many streams of user actions as the number of users. Problems related to undo in collaborative work have been analysed in [72, 3, 89].

In [72] the authors suggest an undo which is neither local nor global, but selective. With the selective undo [12] a user chooses the action to delete. However, his choice cannot be done arbitrarily, because a past action can have had consequences on the following, always past, actions. If an action A precedes an action B and they are independent, in the sense that A does not influence B and the execution order is not important, then the user can select A, can move it from the actual position in the history to the last one and finally can apply the global undo. Moreover, the user has to be sure that no other user is working using the action he is interested in undoing; for this reason, some exclusion mechanism is required.

Exclusion mechanisms are discussed in [3], in which the authors suggest another solution to the undo problem in collaborative work. They suggest the implementation of a local undo, so that any user can know (this happens whenever he knows *how* and *on what* undo is functioning) what he is undoing, with the addition of some exclusion mechanism in order to resolve conflictual situations.

With the exclusion mechanisms at any time, depending on the necessary operation, only one user is able to update the data. Such an exclusion may be controlled by a mechanism of explicit or implicit lock.

If the cooperating users are working with a text editor, with explicit locking, a user that has explicitly locked a portion of a document, performs

his update and then releases the document. During the time in which the document is locked, no other user can access the same resource.

With the implicit lock, the locking of a portion of text is done by the system when a user starts to update it. When the user starts to do something not involving the locked text, or after a timeout, the system releases the lock.

Another exclusion mechanism is represented by roles: each user assumes a role with respect to each object of the document. Such roles may be author, co-author, etc. Depending on the roles, a user can read, write, modify the data.

Finally, the third analysed exclusion mechanism is represented by copying. By doing a copy of the object of interest, any user is working on it as in the single-user environment and, being private, there are no conflicts of concurrent accesses to the same resources. Of course copying should be done with locking, otherwise more users could have also different copies of the same data and may try to modify them concurrently.

Problems typical of collaborative work may be also found in a single user environment: it suffices to think about multi-window systems. A typical example is represented by the word processor Word 5.1. Suppose that a user has opened two files, using two different windows, A and B, in Word 5.1. Suppose that he performs an action in window A and then he simply changes window, making B the working window but without performing any action on it. If he performs undo, the last active action (for a definition of active actions, see Chapter 4) will be undone. In this case, the last action was in A, so the system changes the working window and deletes the last performed action in A. The consequence of this global undo is to disorient the user, since he thinks the undo is acting only on the working window, i.e. the user is thinking in a "local undo" mode.

Multi-user environments are open systems. Their behaviour is extremely difficult to predict, moreover such difficulty is increased by the presence of humans that are unpredictable. Undo, as we will see in the next Chapter, may help in handling and reducing unpredictability. From the user's point of view, a local undo is the most easy and natural kind of undo in collaborative work. So it is not a surprise that the successive release of Word 5.1, Word 6, has been improved also in implementing a local undo instead of the global one.

Chapter 4

Dealing with undo

In this chapter we are going to take into account the unpredictability and non-determinism in interaction. When a user is in control of his dialogue, that is he knows the system state and the system behaviour after any user action, then the interaction is predictable. Undo functions, implicit and explicit, help the user in *handling* non-determinism when he is employing such functions as navigational tools, and/or in *reducing* non-determinism when he is employing them as recovery functions. From the user's point of view, non-determinism in interaction is seen as the risks the user can face while interacting. *Undo*, the explicit undo, is a powerful tool to reduce such risks in interaction. Unfortunately, not all the functions at the application level may be undone; moreover, *Undo* may behave differently in distinct situations. The result of this, is that *Undo* also adds risk in interaction: the more powerful the system, the higher the risk!

4.1 Unpredictability of interaction

Predictions have always been very important to humans. This happened and happens because, in order to predict something, it is necessary to know the object of the prediction, and such a knowledge is, in some sense, a form of control, of power. For a human, a prediction about something is the expression of his power on that thing. It is possible to use such a knowledge in order to exploit the resources of the world to which the knowledge itself is related. It is easier to do predictions when science is involved, as for example in predicting the return of the Halley comet, or when there will be a high or low tide. It is possible to do very precise predictions in a closed

world, controlled by mathematically described physical laws, because these are independent from human behaviour [8]. But when humans are involved in something, it is not possible to do predictions any more. Human behaviour does not follow a physical law. From a human's point of view, we say that something is non-deterministic when it is unpredictable.

Within the computation theory the word non-determinism represents the fact that, starting from the same state, it is possible to perform the same action in more than one way, so reaching different states. The behaviour of a non-deterministic machine may be well modelled by a tree, in which any node represents a state, and more than one arc with the same label (all the arcs with the same label represent the same action) may depart from the same node [78, 47, 60, 49, 14]. This means that, from the same starting state, performing the same operation, it is possible to reach different states following different paths along the tree. Any branch of the tree represents a deterministic computation, but it is not possible to foresee -a priori -which branch will be covered. The unpredicability of a non-deterministic system is a consequence of the fact that at any branching point the system decides which is the way to follow.

The behaviour of an interactive system may also be modelled with a tree, in which any node is a state and any arc is a transition between two states. In this representation, that we can call 'interaction tree', we can neither represent concurrency aspects (as it happens by employing STN) nor cycles, but this choice is necessary in order to do a comparison between an interactive system and a non-deterministic machine.

In interactive systems, at any state the user can perform an action among those belonging to the set of allowable actions for that state. The choice of the action depends on the user's needs, skill, aim, etc. In the 'interaction tree' we can have more than one arc leaving from the same node, any of them representing an allowable action, but each arc has a different label, because the computation is deterministic.

Borrowing "non-determinism" from the theory of computation, we can say that we have non-determinism in HCI because at any node we can have more than one arc leaving and it is not possible to foresee which branch in the tree the interaction will follow, i.e. it is not possible to fully foresee the system behaviour.

Since humans play a role in HCI and they are unpredictable, the resulting interaction is non-deterministic. and, since humans are present within HCI having a double role as that of any partner in a communication (i.e. as a sender when they "say" something, as a receiver when they "obtain"

something) we have a double non- determinism in interaction: one from the computer point of view (when the user is a sender), the other from the user point of view (when the user himself is a receiver).

We have non-determinism, or, as Wegner says, indeterminism [87], from the computer's point of view when the computer is not able to foresee the user succeeding action.

Since interaction is driven by the user, the branch to follow in the 'interaction tree' is not previously chosen, but it is created at any node by the user depending on the feedback, his aim, knowledge, experience, ... Such unpredictability may be reduced if there are good task and user models: in this way, by knowing the task, the user's aim and his knowledge on the system, it is possible to do some prediction on the sequence of user actions, and, consequently, on the computer behaviour. Nevertheless, a user could perform some action not necessarily logically linked with the step foreseen by a task model; sometimes prediction may be right, sometimes not, always because human behaviour does not follow a mathematical law.

From the user's point of view, a definition of non-determinism is slightly more complex. We have non-determinism when a user does not know the system behaviour, so he cannot perform a prediction. In this case the user can gain information on the system behaviour by navigating through the system itself. The choices of which action to perform are generally based on previous knowledge about other systems.

Moreover, there is also non-determinism when a user thinks he knows how the system will behave after his input, but the computer response is different from the expected one. In this case, there are inconsistencies between the user's and system's point of view: the system may appear non-deterministic simply because the user himself has not enough information on the system state and on how it really behaves under the surface of the application software. In both cases, undo functions play a basic role in interaction, in order to allow the user to handle and reduce non-determinism.

Finally, another example of unpredictability in interaction is represented by the fact that in HCI the world is not closed, in the sense that it does not involve only the computer and the user, but, as said in Chapter 2, the system is composed by the computer and all the entities, internal and external to it, which are able to give, and/or react to, external stimuli. This means that we also have to consider printers, elements on the net, ... For this reason, even if the user is following the task model step-by-step, something external to the application (but internal to the system) may happen (as a system crash, problems with printers, net, ...), disallowing prediction.

4.2 Non-determinism from the user's point of view

In the previous Section, it has been said that the knowledge on an object represents for the human a kind of control, of his power on the object itself. While interacting with a computer any user would like to feel in control of his dialogue. In order to do this, Cole, Lansdal and Christie [21] suggested that any user should be able to answer the following questions:

1. *Where am I?* It is important for the user to have information on the state that he has reached. This information is provided by the feedback.

2. *How did I get here?* Changes on the interface after the execution of a particular action allow the user to establish a causality link between that action and the happened changes. This information is also provided by the feedback.

3. *What can I do?* The understanding of the current system state allows the user to know the range of actions that it is possible to perform in that state. This is a direct consequence of the answer to the question *Where am I?*

4. *Where can I go next?* If the user has an aim to pursue but he cannot perform a needed action in the current state, then he needs to change state by reaching a suitable one (reachability). In order to do this, the user has to be able to navigate through the system. This ability is given partially by the feedback and partially by the possibility to perform predictions, basing the last on his previous experience and knowledge about similar systems.

Analysing the answers of the four above questions, we may note that the answer to the first one presupposes a knowledge, at least partial, of the system state; the answer to the fourth question implies a knowledge of the system behaviour, while the answers to the second and third questions require a knowledge of both system state and behaviour. This means that, in order to really feel in control of his dialogue, the user basically needs to know the system state and the system behaviour after any one of his action.

From the user's point of view, the system state is what he can see on the interface of the system with which he is interacting. Actually, the system state also has internal components and the user cannot have all the information on them. However, using some interactive functions, such as *Print*

Preview, *Show* ¶, *Word Count*, *Show Clipboard*, etc., he can find some peculiarities about the system internal structure and state. Also the knowledge on which actions a user can perform at each state is part of the system state.

The user knows the system behaviour if he knows *how* the system executes an action, that is its semantics, and *which state* can be reached after its execution. However, since the visual approach allows the user to have a fruitful interaction without any particular knowledge on the system behaviour under the surface, then we can say that a user has a good knowledge on the system behaviour if he knows simply *which state* can be reached by performing an action, without having a deep knowledge on the semantics of the action itself.

If a user knows which state he has reached, the actions that he can perform and which state he can reach, by perfoming the chosen action, then the system would be totally deterministic, the user could do predictions and feel totally in control of his dialogue. But, since there is not a full knowledge of the system state and of the semantics of the actions, it may happen that some inconsistencies arise between the user prediction and the system behaviour. Some of such inconsistencies may be resolved by employing undo functions, so allowing the user to be more in control of his dialogue.

Moreover, if there is not a full knowledge on the system behaviour and the user is simply navigathing through it, without any specific aim to pursue, then undo functions may be used as navigational tools in order to have information on the system potentialities. Also in this case, by employing undo functions, the user can feel more in control of his dialogue. In the following, in order to differentiate the implicit undo form the explicit one, we will indicate the first as implicit undo, and the last with *Undo*. For uniformity, we will use words in italics starting with a capital letter for any interactive system functions.

4.3 Meaning of undo in interactive systems

In order to express the importance of undo (not only as a recovery function) in interactive systems, a comparison between it and a labyrinth may be useful. In fact a labyrinth is particularly suitable to highlight the exploratory aspects, which differentiate interactive systems with respect to traditional ones. If we imagine an interactive system as a labyrinth, we find ourselves, while walking through it (that is, while interacting), in one of the following situations:

a the user' aim is to reach the centre of the labyrinth; he knows the right way but he makes a wrong tour (slip);

b the user's aim is to reach the centre of the labyrinth; he thinks he knows the right way but he is not able to reach it (mismatch between user and system model);

c the user stopped at a crossing and wants to try a new path. In order avoid walking back (in this case to go back is not intentionally but only a consequence of a wrong choice) he needs to know his previous direction;

d the user has no precise aim, he is simply walking around the labyrinth;

e the user's aim is to reach the centre of the labyrinth, but he does not know where it is. This means that he has to explore the labyrinth in order to reach his aim.

The points **a** and **b** depict that undo functions may be used as recovery functions when the user realises that he has reached a wrong state. The point **c** depicts that the explicit undo may be employed in order to have more information about the system state. Finally, points **d** and **e** depict that undo functions may be used as a navigational tool.

In each of the above mentioned cases, both implicit and explicit undo play a very important role. Actually, not only *Undo* allows the user to repair an error, if it occurs, by reaching a past state (this is the aim of any recovery function), but it allows the user also to handle and reduce non-determinism. In fact, when a user is navigating through a system, exploiting the exploratory aspect, typical of interactive systems, he is enriching his knowledge about the system's behaviour. In this case he *handles* non-determinism. However, his knowledge is related to the external system behaviour; not on the internal system structure or status. Instead, if there is a mismatch between the user's and system's model, then an inexact matching between the interpretation of the user's action from the system's point of view and from the user's point of view occurs: this means that the semantics of the left hand side and right hand side, with respect to the interface I of Figure 2.3, differ. In this case *Undo* may provide more information on the internal system structure, so *reducing* non-determinism.

The main consequence of the above mentioned undo characteristics is that *Undo allows the user to increase his control during his dialogue.*

4.3.1 Undo as a recovery function

After the interpretation of the feedback, the user can realise if the reached state is wrong or not. The reached state may be wrong in two cases:

1. the user *knows* what he wants/needs in order to reach a given state and how to reach it (point **a** in the labyrinth), but he, unwittingly, performs a wrong choice.

2. the user *thinks* he knows what he wants/needs to reach and how to do it (point **b** in the labyrinth), but the reached state differs from the expected one.

In the first case, the user recognises, immediately after the feedback, that the reached state is wrong and, by employing *Undo*, he can cancel the effect of the last performed action(s), so reaching a previous state. In this case, he made what Norman defines as a slip [64]. The slip is an acidental error: for example, when a user writes a wrong character while typing, because his finger is on the wrong key, or when a user clicks on a wrong buttons or icons because accidentally the mouse is not in the right position, then he is making a slip. This situation is easy to recover.

In the second case, the user is performing a mistake, i.e. a semantic error. This means that there is a different semantics of the same action from the user's and system's point of view. The consequence is that there is a mismatch between the user and system model. In this case the *Undo* can provide some information about the internal system structure, so allowing the user to reduce non-determinism.

4.3.2 Undo to have more information on the actual state

While interacting, if the effect of any user action modifies in some way the interface (in the broad sense, considering any kind of feedback, visual, audio, ...), then the user is able to do a link cause-effect. But if there is a breakdown in the interaction, for example if a user stops temporarily his work to have a cup of coffee, then when he comes back, he could not remember how he reached the current state. For example, imagine a user is working with a word processor and in the buffer there is a portion of text that we can call A. Then the user selects the portion of text B. His aim is to do a *Cut* and *Paste* in order to move B from its position to the beginning of the document. Instead he performs a *Delete* on the selected text and then goes away for a cup of coffee. When he comes back, he puts the pointer at the beginning

of the text and performs *Paste*. Result: B is lost (unless selective undo mechanisms are available) and A is pasted. This usually happens when a user does not have or does not use information on how a state has been reached. For a breakdown in interaction, a system function which deletes the effect of the last performed action may help in knowing how the current state has been reached (point **c**). Of course, if the user wants to continue his interaction from the actual state before he used *Undo*, he has to redo the cancelled action.

4.3.3 Undo as a navigational tool

The points **d** and **e** suggest that a navigation through a system may be done without any aim (**d**) or trying to obtain an aim without any knowledge on how to do it (**e**).

The point **d** suggests that an interactive system may be employed by the user simply navigating through it, without any aim. In this way, the feedback following any action in the navigation, allows the user to have more information on the reachable states while interacting. Such information is not about internal states and/or about how the system is functioning under the surface (i.e. information about the actions semantics), but on *what is possible to do* and *what is possible to obtain* with the system. This means that the information acquired from the user is broad, but not deep. Since, in this way, detailed knowledge about system state and semantics is not increased, the user is not reducing non-determinism, but he is *handling* non-determinism to obtain a general idea on the system potentialities.

When a user has an aim to pursue, i.e. a desired state to reach, but does not know how, then, exploiting the exploratory aspects of interactive systems, he can perform an action among all the allowable ones in the actual state. If the reached state is still wrong or inadequate to the user, then he can perform *Undo* cancelling the last performed action and can try another path in the 'interaction tree'. Reasoning in terms of the reachability properties, this means that the user is trying to find a suitable action that, provided as input to the computer, allows him to reach his aim. The situation in this case is very similar to the previous one (**d**), the difference is in the kind and the way of choice of the allowable action that has to be performed, because this time such a choice is influenced by the user's aim.

4.4 Undo reduces non-determinism . . .

In order to understand the system's behaviour, any user needs to know *how* and *on which object* any function acts. The word *how* is related to the action semantics, but we cannot talk about the semantics of each interactive functions. In next Chapters we will focus only on the *Undo* semantics, because this function characterises interactive systems with respect to traditional ones.

The words *on which object* indicate the domain of interest of any action. By applying the visual approach, the user generally does not need to know how the system behaves under the surface of the application software. Unfortunately, all goes well until the user reaches a state that is different from what he forsaw, i.e., until there is a mismatch between the user's and system's model. Such a mismatch may be due to the choice of an unsuitable metaphor to represent visual objects on the interface [57], or to a wrong evaluation of the action semantics. Since, the visual approach implies that the semantics of any system function should be very easy and immediate to understand (see Chapter 2), then a wrong evaluation of the action semantics may be due to a uncomplete knowledge of the object on which the action is working. This mismatch cannot be always resolved by applying *Undo*, because it only cancels the effect of the last performed action(s) and, if the mistake is due to a remote action, then, depending on the implementation of *Undo* (see Chapter 5), it is not always possible to cancel it. Some systems support selective undo which allows the user to cancel individual past actions far back in the history. However, *Undo* can show some aspects of the internal system structure, helping the user in understanding the domain of interest of system functions. *Undo*, by encreasing the user's knowledge of the system's behaviour, may helps the user himself in reducing non-determinism in interaction.

4.4.1 Granularity levels

During any interaction between a user and a computer, it is important to identify the objects which the user can handle, because such objects represent the data of interest, the data which an action can effect. Following the hierarchy proposed in Figure 3.1, we have interaction with operating systems, in which the objects of interest are files, and within application software, in which the objects of interest are contents of files. But the word 'content' is not enough to understand the object which an action effects. So, to better

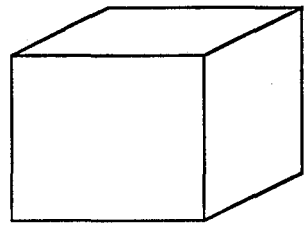

Figure 4.1: This figure is interpreted by the user as a cube, not as a square and 5 segments.

understand the different objects within the application level, we are going to introduce the *granularity levels.*

Within an application software, the levels of granularity are basically two, the lowest and highest, but for each specific application any number of levels between them may be added.

At the lowest granularity level, the data of interest are the elementary components present in the application.

At the highest level, the data of interest are the global content (for example all the text of a document) of files, which are made by string(s) of elementary components.

For example, if we are working with MacDraw II, the elementary components are the geometrical objects which may be chosen in the palette, while the file is made by strings composed by the drawn geometrical objects. In this case the string is not linear, but bidimensional, and, for this reason, it may be seen as a spatial composition of elementary components instead of their linear sequence. In MacDraw II there is also an intermediate level of granularity. Since for the user the spatial composition of some elementary components may have a particular meaning, the system provides a function (*Group*) in such a way that the user himself is able to indicate to the system that the selected components have to be considered as a unique object. Figure 4.1 shows that the drawn picture is considered as a cube by the user, and as a square and five segments by the system. The *Group* function allows both the user and the system to consider the same string with the same granularity.

Some applications may also have an own (or more than one) intermedi-

ate level, besides that one which may be created by the user. For example, working with a word processor, the data of interest at the lowest level are the characters which can be typed by using the keyboard, while at the higher level the data of interest are the content of files, which are made by combinations of characters. The user may create an intermediate level by selecting characters (which, from his point of view, are a portion of a word, or a whole word or a group of words belonging to his text, that is of the written language he is using), in order to inform the system which is the object of the next action. This intermediate level is not persistent and its length in time depends on the successive user action. Besides this intermediate level, there is also another one, this time provided by the system and the user is not explicitly informed about this. While a user is typing, the system considers as a single object the portion of text typed between two successive events. An event is an action, whose name is present in the menu, and may be entered by mouse or keyboard. However, also the time may be an event defining the end of an object. For example, in *Microsoft Word*^{TM}5.1 , if we make a pause of at least 20 sec. while typing, this time is considered an event which defines the end of the object. The next typed character will belong to a new object. Generally the user does not know the structure of the intermediate granularity level, but, since he is using a language (interaction language, and the user does not realise that it is a language) to perform his task (which in this case is based on the grammar of a spoken language) intuitively he associates a meaning to a string of characters, so creating a word of his spoken language. In this way, the granularity conceived by the user is different from the one used by the system. If the user perfoms the *Undo* after typing, its effect is to delete the last written object from the system's point of view, which, as before mentioned, not always corresponds to the definition of object from the user's point of view.

When a user is in a constructing phase during his interaction, that is he is adding elements to his file (for example, he is typing text or drawing pictures), an eventually differences in the granularity between user and system does not effect the following constructive action. But if the user is modifying the file, moving or cancelling some of its content, then inconsistencies due to the different granularity levels may arise. In particular, such inconsistencies are highlighted by the *Undo* which, showing the granularity of the last performed action, increases the user's knowledge about the internal system structure, so reducing non-determinism.

4.5 . . . but Undo adds non-determinism too!

Often, *Undo* is a very powerful tool, allowing the user to repair an error, but sometimes it may add confusion in the user while interacting. How many times have we tried to perform *Undo* and, although present on the interface, such function was not allowable? Or how many times have we performed *Undo* obtaining something different from what we expected? Such confusion is due to the fact that *Undo* is not as ease as it looks; in fact, it is a very complex function and we cannot simply say that it cancels the effect of the last performed action(s). The last action with respect to what? Last from the user's point of view, that is the last action in the interaction (the last user performed action), or from the computer's point of view, that is the last action in the computation, the last action which modified the result? For example, if, working with a word processor a user types a word, then uses the scrollbar and then applies *Undo*, which action will be cancelled? The typing, the scroll or both? The uncomplete knowledge about what and how *Undo* acts, makes its behaviour unpredictable, so adding non-determinism in interaction. In the next Section we will explain what *Undo* acts on, while we will dedicate Chapters 5, 6 and 7 to describe how *Undo* acts.

4.5.1 Different kinds of actions

Undo is a special kind of system function, since its domain of interest is not an object but a command, and its application cancels the effect of the last applied command(s). As said in Chapter 3, explicit undo is available only at the application level, so *Save*, *Exit*, ..., (even if such functions are within an application software, their objects of interest are files, not their contents) and all the other functions at the file level are not undoable in the sense of the explicit undo. However, not even all the functions within the application level may be undone.

Until now, we have talked in general about actions, and such actions could be ordinary commands or *Undo*. Since, as said at the beginning of this Section, *Undo* acts on commands, and not all the commands may be undone, we would like to clarify which commands are undoable and which are not. In order to do this, we have to talk about different command instances and levels.

A command instance may be active, passive or neutral depending on the effect that such a command has on the Result and Display in the sense of the red-PIE (see Chapter 1).

If a command instance modifies both Result and Display, then such instance is said active; if it modifies only the Display but not the Result, then such instance is said passive; finally, if it modifies neither Display nor Result, then such instance is said neutral. More formally, given the red-PIE $\mathcal{P} = < P, I, E, result, display >$, assuming that the system, starting from the state s_a, by executing the command c, reaches the state s_c, that is $doit(s_a, c) = s_c$, we have the following definitions:

Definition 4.1 *A command instance* **c** *is said* **active** *if:*

1. $s_a \neq s_c$;
2. $result(proj(s_a)) \neq result(proj(s_c))$;
3. $display(proj(s_a)) \neq display(proj(s_c))$.

Definition 4.2 *A command instance* **c** *is said* **passive** *if:*

1. $s_a \neq s_c$;
2. $result(proj(s_a)) = result(proj(s_c))$;
3. $display(proj(s_a)) \neq display(proj(s_c))$.

Definition 4.3 *A command instance* **c** *is said* **neutral** *if*

1. $s_a = s_c$;
2. $result(proj(s_a)) = result(proj(s_c))$;
3. $display(proj(s_a)) = display(proj(s_c))$.

The same command, applied in different state, may have different instances. For example, suppose that a user, while working with a word processor, selects a word, performs a *Cut* and then performs a *Paste*. In this case *Paste* has an active instance, since such command modifies both Display and Result. If the user selects the just pasted word and performs again a *Paste*, then its instance is neutral, since neither Result nor Display have been changed.

At this point we can define a command as active [76], passive [39, 26] or neutral as following:

Definition 4.4 *A command $c \in C$ is* **active** *if there exists at least one occurrence in which c has an active instance.*

Definition 4.5 *A command $c \in C$ is* **passive** *if there exists at least one occurrence in which c has a passive instance; c may have a neutral instance, but never an active one.*

Definition 4.6 *A command $c \in C$ is* **neutral** *if there exists at least one occurrence in which c has a neutral instance; never c can have an active or passive instance.*

Working in *Microsoft Word*^{TM}5.1, example of active commands are *Cut*, *Paste*, *Replace*, *Clear*, etc.; example of passive commands are the scrollbar, the cursor movement, *Outline*, *Page Layout*, etc.; examples of neutral command are *Word Count*, etc.

In the most common application software, *Undo* acts only at the level of the active commands. If an active comman is followed by passive or neutral commands, they are simply skipped. For example, in *Microsoft Word*^{TM}5.1, if a user, types a word, then, with the scrollbar, changes the display and then performs *Undo*, the effect of the *Undo* is to skip the scrollbar and to recover the situation in which the text was before the user typed the cancelled word. It is logical to understand why *Undo* does not act on neutral commands, because the effect of such commands is not to modify result or display but to provide information to the user. In some way, it is logical also to understand why *Undo* does not act on passive commands, because cursor movements, scrollbar, etc. are easy to undone with implicit undo. Nevertheless, the passive commands are not always skipped. Consider, for example, in *Microsoft Word*^{TM}5.1 the functions which allow the user to change the view of the document (as *Outline*, *Page Layout*). Since such functions modify the display but not the result, they are passive. In any of these modes, the user can edit, scroll and undo active actions, as in the normal mode. While the user is working within one of these modes, *Undo* operates uniformly and consistently. However, when swhitching between them the situation is different. If a user, while working in the normal mode, enters an active command, and then changes the view of the document applying *Outline*, he cannot perform *Undo* because the last passive command is not skipped. Conversely, if the user changes view with the *Page Layout*, then, performing *Undo*, the last passive command is skipped and the last active one is undone. The situation is the same also when moving from the Page Layout or Outline mode to the Normal one. Moving from the Page Layout to the Normal, the last active action done in Page Layout may be undone, since the change of the view is seen as a passive action. Moving from the

Outline to the Normal, the last active action done in Normal cannot be undone. Such problems have been resolved in *Microsoft Word*^{TM}6.0.

There are special kinds of active commands as *Replace*, *Character*, etc., which allows the user to do more than one modification at the same time. For the execution of such commands, the user is helped by a dialogue box and performing *Undo* all the modification are undone at the same time. For example, when a user closes the *Replace* dialogue box, after having applied it, and performs *Undo*, then all the modification done with the *Replace All* are undone at the same time, because the system clumps together the sequences of active actions requested within the dialogue box and treats them as one. The user cannot perform an explicit undo, but only an implicit one, while he is in a dialogue box. The opening of a dialogue box under the menu item Edit is a passive command and while it is opened, the user can perform the *Undo* of the last active entered command in the text. Conversely, when any dialogue box is opened from the menu item Format, then it is not possible to do any operation until the submenu is closed.

4.6 Undo and risky interaction

In the previous sections, the importance of undo functions (implicit and explicit) has been introduced in order to draw importance on how the user can feel more in control of his dialogue. The implicit undo (as when moving an icon from a window A to a window B and vice versa) relies on proportionate effort: small actions give rise to small effects (remind that big actions, as to remove a file, have neither implicit nor explicit undo). But what we expect when interacting with a computer, is to be able to do much very quickly, that is to obtain large effects from small actions. Usually, such an increase of power is at the application level, but it does not always implies a power in interaction too! Really, by increasing the application power, also the risk in interaction increases. Furthermore, an explicit undo may be extremely useful to reduce such a risk [35]. But what is this risk? The first kind of risk may be due to the presence of user's errors (slips) while interacting, so the *Undo* can help the user in recovering from errors. Moreover, the other risk is due to the mismatch between the user's and system's point of view on the same action.

Not always the presence of *Undo* assures a reliable interaction: in fact, it often can make a system more risky. In the previous Section we gave an example of risky *Undo*, when we said that in *Microsoft Word*^{TM}5.1

not all the passive actions are skipped when applying *Undo*. This means that the behaviour of this system function may add non-determinism in interaction when it is slightly inconsistent. But does it matter? For 99% of the time *Undo* works, this is better than systems where *Undo* was either absent or frequently did not work. Or is it? If *Undo* works almost all the time, then users may become used to it. Indeed, one of the advantages of having *Undo* is that users can take a more exploratory approach to their interaction, trying out possible courses of action, but then retracing their steps if negative results occur, i.e. their behaviour become more risky. But if they become more risky then they are more likely to do things that need *Undo*, and if *Undo* does not work uniformly then they will have problems which would never have occurred if they had been more careful. So although a 99% effective *Undo* may not be better than having no *Undo* at all, there will be errors that occur when you have *Undo* which would not have happened if you had never had *Undo* at all!

Chapter 5

Reflections on Undo and Redo

In this chapter we are going to take into account characteristics of the *Undo/Redo* functions in interactive systems. We identify the domain of interest of *Undo* as the command history. Actually, we could have two classes of *Undo* function, one self-applicable and the other not self-applicable. In the first case, *Undo* belongs to its domain of interest and *Undo* of *Undo* is it is used as the *Redo* function. In the second case *Undo* does not belong to its domain of interest and the *Undo* of *Undo* is used as a backtrack tool. Besides this classification, some systems allow also to perform *Undo* not only on the last performed action, but also on a block of n actions. The size of the block of actions to be undone is decided by the user or by the system, depending on the implementation. Two semi-formal definitions of *Undo* and a taxonomy of interactive systems, based on the *Undo* granularity and repetition, are next proposed. Moreover, we will also introduce the *Redo* function. Its application domain is not the *Undo*, but the undone actions. The *Redo* is not exactly the inverse of *Undo*, there is, instead, an intrinsic causality dependence with the *Undo*. Two semi-formal definitions of *Redo* and a taxonomy of interactive systems, based on the *Redo* granularity, repetition and the link with *Undo*, are proposed. Finally, a discussion on reflexivity aspects, reachability properties and commitment points, stresses the fact that by increasing the power of the *Undo/Redo* mechanism the risk in interaction increases.

5.1 The world around Undo

The issue of *Undo* in user interfaces has been studied by several authors over many years (e.g. [4, 45, 88, 84, 62]). This has included both work aimed at understanding the problem, and work on implementation structures. Despite this, experiments have shown that experienced users of Microsoft Word,which has a relatively simple and easy to use *Undo* function, still have great difficulty in working out what *Undo* will do in some contexts [66]. Is this because we do not still have a clear idea of what *Undo* should do, or is it simply that *Undo* is intrinsically complex?

This is not simply a matter of theoretical interest. At the time of the earlier formulations of *Undo*, the users of most interactive systems were either experts, or at least computer literate. Even if the users of a system with complex *Undo* mechanisms, such as Emacs [83], did not fully understand its semantics, at least they were not too intimidated by its often erratic behaviour. Now, sophisticated multi-step *Undo* is available on standard office systems such as Microsoft Word 6, and indeed the ability to undo with ease (not necessarily with an *Undo* command) is seen as one of the positive key features of the direct manipulation paradigm [81].

But what does *Undo* exactly achieve? Since most systems allow the user to reach only the previous state, one could think that the *Undo* is a system function which allows the user to delete only the previous action. Some systems do not allow to perform the *Undo* of the *Undo* , some others allow it; in this last case, if it is possible to go back along the past history, the *Undo* of the *Undo* may be used as a backtracking tool, otherwise the system may oscillate between two states. Moreover, some systems allow not only to reach the previous state, but also any one in the past history.

From the above considerations, we can argue that the world around the *Undo* is very confused. Given the range of interpretations of *Undo* in different applications, it is clear that there is not yet a common understanding or definition. Its weak definition, including all the above mentioned examples, consider the *Undo* as a system function which allows the user to reach any state in the past history. This means that the *Undo* may be seen as a special case of reachability [29]. In fact, if, in a system, the reachability property holds, it is possible to reach, starting from any system state, any other state, both the ones previously reached (present in the past actions history) or others not yet reached (possible future actions). Therefore, reachability allows the user to move in both directions of the action history, past and future. The explicit *Undo* is a special case of reachability, in the sense that

it allows the user to move only in one direction of the action history, the past one.

Among all the functions which a user can perform while interacting with a computer, *Undo*, as we will explain in the following sections, is one of the most complex and its behaviour differs from any other system function. Particularly, after an *Undo* is performed, the user may find inconsistent situation due to the different granularity of the handled data or to the different kind of performed actions. This inconsistency arises because *Undo* shows something of the internal functioning of the system, while the user does not know anything about it: if the new information he receives is in contrast with what he knows or thinks, he has problems of inconsistency.

5.2 The script model

Different formal models have been proposed in the literature in order to describe *Undo* in interactive systems [4, 86, 84]. In [4] the authors propose a 'script' model of *Undo*, which we will refer to as the ACS model. This is based on three streams of actions: the user history, the active script and the pending script. The user history is simply a list storing all the user's actions. Sequences of commands produce scripts; there is an immediate mapping between each script and a state as the object visualised on the screen. The active script is the list of the user's entered commands. The pending script is the list of the commands deleted in the active script by the *Undo*.

If, for example, we are working with a text editor and we enter these actions in the user history 'type(hi), type(everybody), *Undo*', we will have this sequence:

User History type(hi)

Active Script $< type(hi) >$

State hi

Pending Script <>

User History type(hi), type(everybody)

Active Script $< type(hi), type(everybody) >$

State hi everybody

Pending Script <>

User History type(hi), type(everybody), *Undo*

Active Script < *type*(*hi*) >

State hi

Pending Script < *type*(*everybody*) >

This example shows that in any interaction there are two kinds of history, one storing any user action which the user can only increase by performing actions, the other storying the active script and the user modifcations, adding or deleting commands. The role of the pending script is to keep trace of the last undone action. If a *Redo* follows an *Undo*, then the action present in the pending script is moved from its position and is added to the active script, while the *Redo* function is added to the user history. The *Redo* case is shown in the following example:

User History type(hi), type(everybody), *Undo*, *Redo*

Active Script < *type*(*hi*), *type*(*everybody*) >

State hi everybody

Pending Script <>

5.3 Is Undo part of the commands' history?

The difficulties in dealing with *Undo* arise for its complex nature and structure. In order to understand how this function acts, it is better to note the differences between it and the ordinary commands. Broadly speaking, we can divide all the user's actions in two classes: the first is made by all the functions which are strictly related to the user's task; the second is made by function(s) which allow the user to modify the past interaction [88], that is the *Undo* functions (*Undo*, *Redo* or browse).

Indicating with A the set of user's actions, we have that $A = (C \cup U)^*$, where C is the set of allowable commands and U is the set of *Undo* functions,

including all different kinds of *Undo* (backtrack, flip, etc.), *Redo*, and/or browsing functions. In the case of a single *Undo*, without any *Redo* or browsing, this simplifies to $A = (C \cup \{Undo\})^*$.

We will use H^a to denote the set of sequences (or histories) of actions ($H^a = A^*$), and H for the set of commands histories ($H = C^*$). That is, H^a corresponds to the 'user history' part of the ACS model, and H corresponds to the commands in the 'active script' or, in other words, the commands issued to the system as if there were no *Undo*. In the sequel, when we will talk about the *Undo* of a command, we will intend *Undo* of an undoable command.

The subdivision of the user actions into commands and *Undo*, is naturally generated by the different user aim:

command The user's aim is to modify an object.

Undo The user's aim is to delete a modification, the effect of a command on an object. In other words to modify the interaction itself.

Starting from the above distinction, it is possible to provide a functional description of both an ordinary command and the *Undo*.

An ordinary command $c \in C$ is a function that modifies an object of interest, that is, an object directly related to the task the user is performing. Such objects are those that are in the state of the system, even if we ignore *Undo*; that is S. The primary purpose of commands is to act on S. This can be modelled using the *doit* function (see Chapter 1). We have that $doit(s, c) = s'$, that is, performing the command c we can switch from state s to s'. Now, s' is a new state, typically distinct from all the previous states, and both s and s' belong to the set of states S. In the ACS model, *doit* corresponds to the result of the natural mapping between the active script ant the actual situation of the document, ignoring the history. Note that this *doit* function only tells us about the effect of ordinary commands on the state of the system *without Undo*. They will also have some effect on the rest of S^a determining the complete behaviour of the system.

Turning now to the *Undo*, the object of interest of *Undo* depends on which definition we consider. However, its data of interest are not the same objects on which the commands act. Instead, such data may be commands or actions, that is, if we consider *Undo* without *Redo*, the application domain may be the command history (H) or the action history (H^a).

In the first case, *Undo* is definded by a function $U : H \to H$ acting on previous commands to reverse their effect. In this case, the effect of *Undo*

itself cannot be reversed, since it does not belong to its domain definition. Effectively, all past undos are forgotten, except for their effect in having reversed previous actions. Such a system cannot have *Redo* function; *Undo* is not self applicable and *Undo* of *Undo* acts as a pure backtracking tool.

Alternatively, the domain of interest of *Undo* may be the complete action history (from H^a). In this case, *Undo* can be definded by a function like $U : H^a \rightarrow H^a$. That is, the system regards *Undo* as a command. However, it is not natural and, in fact, there must be a different system behaviour when performing *Undo* after a command or after a previous *Undo*. This is exactly what we see in all systems with *Undo*. When the previous action has been a command, then we expect *Undo* to act upon it by reversing its effect. In this case, we have $doit^a(s, Undo) = s'$, where s' is not exactly a previous state, but in some sense an equivalent one, because the system keeps track of the *Undo* activity (for example using a boolean variable or the pending script).

When performing *Undo* of *Undo* , different systems may have different behaviour: some of them simply do not allow it, others consider *Undo* of *Undo* as the *Redo* function. We will consider *Redo* later, but the former case, of simple single step *Undo*, can be informally described as:

$$U(h \frown a) = \begin{cases} h & \text{if } a \in C \\ not\ allowed & \text{otherwise} \end{cases}$$

The peculiarity of *Undo*, is that it is not a command but a meta-command, and, being so, its structure is quite different from the one of the ordinary commands. When using *Undo* as a command, it is self-applicable (*Undo* of *Undo* as *Redo*) and some aspects of this reflexive structure are revealed to the user, giving rise to problems of inconsistency (as said in chapter 4) and even apparent randomness, especially if the user is expecting a different kind of *Undo* behaviour. Moreover, since the domain of interest of *Undo* is an action or command history, when using *Undo*, the user is not simply interacting, but instead he is *interacting with interaction.*

5.4 Single-action and multiple-action

In the previous Section we began to classify different kinds of *Undo* based on whether *Undo* is self-applicable or not. We have not explicitely considered the *Redo* function. Moreover, we have also been intentionally vague as to the *scope* of *Undo*, which varies markedly between specific *Undo* systems. This

Granularity / Repetition	Single action	Multiple actior
Single undo	(i) Undo only of the last command. Undo of undo is not allowed	(iii) Undo of a block of actions. Undo of undo is not allowed
Multiple undc	(ii) Undo only the last command. Undo of undo as backtracking	(iv) Undo a block of actions. Undo of undo as backtracking

Figure 5.1: A taxonomy of *Undo* function. The rows represent the granularity, the columns represent the repetition of *Undo*.

scope has two main aspects. Firstly, the number of times that *Undo* can be applied: single-undo or multiple-undo, where by single-undo we mean the ability to apply *Undo* only once, while by multiple-undo we mean the ability to apply *Undo* successively. Secondly, the number of actions that may be undone at one step: many systems allow only a single action to be undone at a time, but other systems allow multiple actions to be undone at each *Undo* step. The latter is a classification based on *Undo* granularity, that is, how many actions may be undone at any step: one or many. In the case of the multiple-action *Undo*, the user 'decides' how many actions to undo at one step. In other words, at a low level the system determines the granularity of undoable actions, whereas the user determines the granularity in terms of the number of such actions to be undone. Unfortunately, because of the way *Undo* 'digs' beneath the surface of the system implementation, several different sorts of granularity are important and we have to ignore some in order to understand others.

Figure 5.1 summarises a classification based on the above two distinctions: single/multiple *Undo* and single/multiple actions. In it we identify four classes of systems:

(i) *single-undo/single-action*, where it is possible to apply *Undo* only on the last performed action and the *Undo* of *Undo* is not allowed;

(ii) *multiple-undo/single-action*, where it is possible to apply *Undo* on the last entered command and the *Undo* of *Undo* is used as a backtracking tool;

(iii) *single-undo/multiple-action*, where it is possible to apply *Undo* only on the last block of n actions and the *Undo* of *Undo* is not allowed;

(iv) *multiple-undo/multiple-action*, where it is possible to apply *Undo* on a block of n commands and the *Undo* of *Undo* is used as a backtracking tool.

It is easy to find examples of systems in three of these classes: for instance, the standard single-action *Undo* (i) is found in many spreadsheets, graphic packages and word processors; (ii) describes the behaviour of the *Back* command in HyperCard; and multiple-action/multiple-undo (iv) is supported via a pull down menu in Word 6. However, systems of type (iii), at the top right of the diagram, seem to be absent. Why is this so? In principle, it would be possible to produce such a system, but it would not be convenient to do so. We will see why in a moment.

One factor that differs in these kinds of *Undo* is the amount of information they have to store in order to be able to perform an *Undo*. In the case of (i) single-undo/single-action, only the current state and previous state need to be stored. Every active command *commits* the previous one, in the sense that they cannot longer be undone. This is a backward commitment point, as it limits the amount the system can go 'backwards' in time to previous states. In contrast, (ii) and (iv), both of which allow multiple-undo, have no backward commitment points; it is always possible to go as far back as you like. Among other things, this means that systems of type (ii) and (iv) must both store similar amounts of history information.

The presence of backward commitment points are bad news for the user, as they limit the possibilities for recovery. However, they are good news for the developer, as a commitment point limits the amount the system has to store and hence the cost of the *Undo*. In the extreme, storing *everything* can be very expensive (even when implemented carefully using 'deltas'[1]),

[1] 'Delta' is a term used in version control systems. Instead of storing every version of a document individually, only some entire copies are kept (often the oldest or most current version) and in addition information is stored to describe differences between versions of

so many systems have slightly weaker forms of (ii) or (iv) where there is a limit on the number of commands that can be undone (e.g. one hundred commands in Word 6), or on the total resources used to store history information (e.g. Emacs, which has a large byte count limit). However, we will regard these as effectively falling into the relevant category, just as we regard a spreadsheet as being able to handle arbitrarily large sheets even there is some resource limitation.

Looking at the concept of backward commitment points, it is clear why it is unusual to find systems of type (iii). Such a system would have no backward commitment point so long as only ordinary commands were used. Similarly, like (ii) or (iv), it would, in principle, have to remember the complete history of the interaction. However, after a single n action *Undo*, it would no longer be possible to go back beyond those n actions. That is, the action of doing an *Undo* would establish a backward commitment point. Such a system would have all the disadvantages of (ii) or (iv) in terms of potential cost of mantaining history information, while making things worse for the user by establishing backward commitment points, reducing the possibility of recovery. Not surprisingly type (iii) systems are rare!

The relationship between (ii) and (iv) is also rather interesting. To see this, let's consider two informal definitions of *Undo*:

Definition 5.1 Undo *is a system function which allows the user to reach* the *previous state.*

Definition 5.2 Undo *is a system function which allows the user to reach* any *previous state.*

Neither of the above definitions of *Undo* allows selective undo be considered exactly an *Undo*. In fact, after its application, the reached state may be not the one previously reached. For example, if a user performs the sequence of actions 'a, b, c', and realises that 'b' is wrong, then he can select and delete it. But what he obtains is 'a, c'', because the execution of 'b' may in some way have influenced the following action. And the sequence 'a, c'' may not be a part of the previous history.

Conversely, Definition 5.1 clearly covers type (i) *Undo* (single-undo/single-action) and type (iv) clearly falls under Definition 5.2. What about (ii)? On

the document - the deltas. Forward deltas enable a more up-to-date version to be recreated from an older one, backward deltas allow older version to be recreated from newer ones.

the one hand, a single *Undo* always turns back to *the* previous state. However, because the user can apply *Undo* repeatedly it is possible to go back to *any* previous state: one can always get the effect of a single n-step *Undo* by doing n single-step undos. So, the difference could be seen as one of *task migration* [32]; that is, the same objective can be reached either by the user or the system. Furthermore, given that the mapping between physical actions and logical actions is rather a matter of taste, one could even regard the user pressing the *Undo* button n times as being equivalent to a single logical n-step *Undo* action!

So, to some extent, (ii) and (iv) give the user equivalent power, but with a different user interface. In fact, for n-step *Undo* the user interface issue is particularly complex. When an action is performed one wants to have some idea of what the action is going to do (predictability). Similarly, when one performs an *Undo*, one would like some idea of what will happen. For single-step *Undo* this can be difficult, as shown by the study of Wright et al. [66]. However, for n-step *Undo* things are much more difficult. Even if one has a very clear idea of the granularity of each command, can one remember just how many commands back lies the state one is trying to return to? For class (ii) one can simply go back until one notices that he/she is in the right case, but for n-step *Undo* it is essential that the system gives some means to determine how many steps to go back. For example, in Word 6 the previous actions are presented in the *Undo* menu.

Having an n-step *Undo*, whether supplied by the system (iv) or by multiple commands (ii), makes it also possible to go back too far by accident. For case (ii), one would probably notice and, at worst, *Undo* one step too many. In case (iv) the potential damage is greater. For multiple undos, *Redo* is not a luxury, but a necessity.

5.5 Adding Redo

The raison d'etre of *Undo* is the user's need for a function that allows him to reverse the effect of a command, recovering a past situation. This is needed when the command has been performed as an error so that an undesirable state has been reached. What happens if the user realises that *Undo* has been performed in error and has itself resulted in an undesirable state? The answer to this question has been the raison d'etre of the *Redo* function.

It is common to consider *Redo* as the inverse of *Undo*. Indeed, this may be the semantics of *Redo* when *Undo* is applied on a single action. But

the meaning of *Redo* is less clear in the case of multiple-undo. Is its effect the reconstruction of the last undone action, or of all the deleted history? And what about the effect of *Redo* when the last *Undo* has deleted a block of actions? Does it recover the whole block or only the last action in the block? Moreover, not only what *Redo* does is unclear, but there is also a complex dependence between it and *Undo*. In the case of single-undo/single-action, *Undo* is considered the inverse of the undone command and *Redo* the inverse of *Undo*. But in any semigroup (the set of command is a semigroup [29]) if the inverse exists, then the inverse of the inverse of an element is the element itself. This means that the inverse of Undo *(type('x'))* is *type('x')*. But *Redo* is not identical to *type('x')*, it is just that when performed at a *particular point of the history*, it has the same semantics. This has two consequences. Firstly, we could consider *Redo* as a sort of super syntactic sugar. In principle, the user could simply repeat the undone command; *Redo* just makes this easier (possibly substantially easier). We could say that the domain of interest of *Redo* is not so much *Undo* itself as the *undone action(s)*. Secondly, like *Undo*, we have to consider at what level we expect *Redo* to reverse the effect of *Undo*. Certainly *Redo* is not *exactly* the inverse of *Undo*!

After saying what *Redo is not*, we need to progress towards some definition of what it *is*, or at least, as we did with *Undo*, explore the range of options.

When considering *Undo*, four major issues arose: reflexivity, granularity (single or multiple action), repetition (single or multiple *Undo*) and the idea of commitment points. Each of these has parallels for *Redo*, and in addition the properties of *Redo* are linked to those of *Undo*. Although *Redo* may not be a simple inverse of *Undo*, it is intimately connected. We will see such dependence when considering the granularity of *Redo*, and also that there is an intrinsic dependence of causality that determines whether *Redo* is meaningful.

5.5.1 Causality

Just as you cannot think of *Undo* without considering what has been *done*, you cannot consider *Redo* without something having been *undone*. This gives rise to the most basic property of *Redo*:

causal dependence: in order to perform a *Redo*, *Undo* must have been performed.

This appears too obvious to bother stating, but serves to highlight the reflexive nature of *Redo*. With *Undo*, we had to consider whether the principal domain of definition is the ordinary command history, or the action history itself. With *Redo*, we move up a level: is it (i) simply the command history, (ii) the history with *Undo* commands, or (iii) does it also know about its own role in the interaction? The causality condition would imply at least some knowledge at level (ii). So, its *effect* may be simply in terms of the undone actions, but it must at least know that they have been undone.

5.5.2 Granularity and repetition

Redo, like *Undo*, may be applied to single or multiple actions. However, there is the additional issue of the extent tied to the granularity of the *Undo* command. We refer to such a linkage as *granularity dependence*. In addition to there being one or more candidate *undone* commands to *Redo*, these may have arisen because of one or more actual *Undo* commands. This gives rise to five kinds of potential *Redo* granularity:

(a) *Redo* of last undone action

(b) *Redo* of some number of undone actions

(c) *Redo* of all the actions undone by the last *Undo* command

(d) *Redo* of all the actions undone by some number of *Undo* commands

(e) *Redo* of all undone actions (up to the last non-undo command)

Within this list, (c) and (d) exhibit granularity dependence, whereas (a), (b) and (e) only exhibit causal dependence (they *Redo undone* actions).

The last (e) corresponds to a sort of escape, which reverses the effect of an entire sequence of undos. Similar escapes occur at the ordinary *Undo* level; for example, many systems have a 'revert' menu option, which allows you to restore a document to the last saved version. Such escapes are themselves a sort of *Undo* operation and are often considered in the same context [88]. Given that the effect of *Undo* can be so confusing, such an escape from an *Undo* dialogue may well be a good idea!

We can look at each of the *Undo* categories on Figure 5.1 and see how they interact with these kinds of *Redo* granularity. Recall that class (iii), single-undo/multiple-action, was deemed an unreasonable alternative, so we will only consider the other three cases.

(i) *single-undo/single-action*: In this case, there can only ever be one undone action and one (effective) *Undo*, so all five *Redo* categories collapse into one.

(ii) *multiple-undo/single-action*: In this case undone commands and *Undo* commands are in a one-to-one correspondence, so (a)=(c) and (b)=(d). However, categories (b) and (d) look weird. If the system is going to allow single *Redo* commands to have non-singular effects, why not allow this for *Undo*?

(iv) *multiple-undo/multiple-action*: In this case, (a) is the weird option. If you can *Undo* groups of actions, why only allow single *Redo* steps? The same argument could be said to hold for (c) with respect to (d), but perhaps, given the different semantic level, one could argue that in some systems (c) may be more comprehensible than (d).

As with *Undo*, we find that the granularity of *Redo* interacts strongly with the possibility of repeated *Redos*, but in addition it also interacts with the classes of *Undo*. We can consider the remaining categories above and see which make sense when we consider single and multiple *Redo*.

With case (i), multiple-redo is meaningless (only one thing to *Redo*!), leaving us a single category of *Redo*, flip-undo, where the *Undo* and *Redo* toggle between two states. As only one of *Undo* or *Redo* is possible at any time, the same button or menu position is used for each, leading to the apparent situation where *Undo* is self-applicable. However, as we saw earlier, this '*Undo* of *Undo* is *Redo*' situation is never quite uniform between *Undo* and other commands.

For both cases (ii) and (iv), the 'escape' *Redo* can only be invoked (as a *Redo*) a single time (although of course it might toggle, like flip-undo, undoing the *Redo*!). Would one want such a *Redo* in these circumstances? It might be argued on efficiency grounds: a system may store only *backward deltas*; that is, information sufficient to *Undo* commands, but not *Redo* them. During a cycle of undoing, the system needs only to store the last not-undone state and the current state: the escape *Redo* would simply jump back to this last not-undone state. However, although this is credible, the extra expense of two-way deltas over and above one-way deltas is not enormous and so it is likely that a *Redo* of the 'escape' form would only be supplied in addition to more incremental *Redo*.

In case (ii), we dismissed options (b) and (d), leaving only *Redo* granularity (a/c) to consider. For reasons similar to those that ruled out *Undo*

of class (iii), we can also see that allowing only a single *Redo* of granularity (a/c) would not be convenient. If we allow repeated *Undo*s, we have to have all the expense of machinery and memory to store lots of states, so why not allow multiple invocations of *Redo* also? That is, we should only have options (a/c) with multiple *Redo*, where each *Redo* reconstructs more and more of the undone history of commands.

Finally, in case (iv), we have a similar story. Options (b), (c) and (d) only make sense for multiple *Redo*, where they perform a similar job reconstructing the command history.

Figure 5.2 summarises this taxonomy. Note again the 'diagonal' emphasis of the table: granularity and repetition correlate, both within the operations of *Undo* and *Redo*, and between them. If you are going to go to all the trouble of storing many history information you might as well use it!

As we did for *Undo*, we can summarise this in two informal alternative definitions:

Definition 5.3 Redo *is a system function which allows the user to recover the past state removed by* the *previous* Undo.

Definition 5.4 Redo *is a system function that allows the user to recover a past state removed by* any *previous* Undo.

Definition 5.3 corresponds to flip-undo. As with *Undo*, there is a design choice between achieving Definition 5.4 by the user doing series of redos (cases ii.a/c and iv.c), or with a single large granularity *Redo* (cases iv.b and iv.d). Finally, the difference between (iv.b) and (iv.d) is in the interpretation of 'a past state' in Definition 5.4, whether it is '*the* past state removed by any previous *Undo*' or '*any* past state removed by any previous *Undo*'.

5.5.3 Reflexivity

As we saw in Section 5.5.1, there is an inevitable reflexivity in the nature of *Redo* – it cannot exist without reference to the previous occurrence of *Undo*. In the ACS model this is captured in the state of the 'pending script'; however, for some kinds of *Redo* (i.a) this is overkill, for others (iv.c/d) insufficient. The active script and pending script contain only the ordinary commands so representing a low level of reflexivity: looking at the interaction with the underlying application. The more complex cases require the 'pending script' to record aspects of the complete action history – *Undo*/*Redo* are reflecting on their own behaviour.

Undo \ Redo	Single redo	Multiple redo
(i) Single-undo Single-action	(a) = (b) = (c) = (d) = (e) Redo only of the last undone action	Only one command to redo
(ii) Multiple-undo Single-action	(a) = (c) Redo only of the last undo of a sequence (e) Redo as an 'escape'	(a) = (c) Redo of the last performed undo. Repeated redo is used to reconstruct the history
(iv) Multiple-undo Multiple-action	(b/c/d) Single redo of a block of undone commands. Size may be different from the last undo. (e) Redo as an 'escape'	(b/c/d) Redo of a block of undone command. The size may be different from the last undo. Repeated redo used to reconstruct the history

Figure 5.2: A taxonomy of *Redo* function. The rows represent the granularity; the columns represent the kind of *Undo* which may precede Redo.

For systems where the *Undo* can be described purely in terms of the pending script, the *Redo* and *Undo* operations can be regarded as having a domain of the form $H \times H$, (active script $\times$ pending script). The flip-*Undo* is a degenerate example, as the pending script never has more than one command (only one level of *Undo* is allowed), and the causal dependence is captured by the fact that the pending script is not empty only if there has been a previous *Undo*. Based on this, it is possible to fully describe flip-undo using three rules:

1. ordinary command – add it to the active script and empty the pending script

2. *Undo* – if the pending script is empty remove the last command from the active script and put it in the pending script

3. *Redo* – if the pending script is non empty remove the command from the pending script and add it to the active script

Since the last two of these rules are disjoint a single button (or menu option) can be used. Although this is a valid description of the behaviour, it is not how any such system is actually implemented – one wouldn't bother to store the whole active script and then never use it! Indeed, even for the formal specification, we will use just two copies of the state: current state and past state – similar to the single-step *Undo*. With such a representation, both *Undo* and *Redo* simply swap the two states – identical! The system does not need to know whether it is doing an *Undo* or a *Redo*, the difference is in the user's interpretation of the effect. This is closer to the way it would be implemented.

The most common and straightforward kind of multi-step *Undo* / *Redo* can also be described using the basic ACS pending script. This is the policy found in Microsoft Word 6 and in the history list of Netscape Navigator. In these systems you can *Undo* any number of commands one by one, or even *Undo* several commands at once, using a menu. The behaviour can be described in a similar manner fot the flip- undo:

1. ordinary command – add it to the active script and empty the pending script

2. *Undo* n – remove the last n commands from the active script and add them to the pending script

3. *Redo n* – remove the last n commands from the pending script and add them to the active script

Notice that (as we saw with single-step and backtrack *Undo*) the description is *simpler* (no conditions on the pending script) because it is more uniform, even though it is far more costly to implement. Word 6 and Netscape use different interface representation metaphors: in Word 6 the user has separate *Undo* and *Redo* menus, which exactly correspond to the active and pending scripts, whereas in Netscape there is a single 'Go' menu with a tick against the currently displayed page. The Word 6 menus show commands (e.g. 'typing'), whereas in Navigator the items in the menu are pages, which correspond to states. The latter difference is a direct consequence of the more identifiable nature of the web browser state (a WWW page). Note also that the Netscape interface suggests a model that, rather than having two scripts, has just one script with a pointer $H \times Ptr$. This is equivalent to the $H \times H$ representation, but is in some ways more flexible (as we will see below).

Not all *Undo* systems can be described using a simple pending script. Systems of type (iv.c/d) need to have some record of how many commands were undone by a previous *Undo* in order to *Redo* them. The raw pending script merely records the list of undone commands, such systems need a pending script that itself contains *Undo* commands! In fact, it is easier to think in terms of a pointer into the complete action history; that is, *Undo /Redo* acting on a domain of the form $H^a \times Ptr$. All commands add something to the end of the history. Ordinary commands add themselves and set the pointer to the end. The *Undo /Redo* command takes the action currently pointed to, adds the inverse of the action to the end of the list and moves the pointer back one. Whether the *Undo /Redo* command is regarded as *Undo* or *Redo* is dependent on what sort of command is pointed to, and depending on how the inversions of commands are represented, the difference may be one of interpretation, rather than of different behaviour within the application. This sort of strong reflexivity sounds quite complex, and indeed in GNU emacs, where it is used, no amount of experimentation seems to be able to uncover the rule! However, exactly the same rule is used in HyperCard's *Back* menu function (one of its two forms of history), and it seems less confusing there. This form of *Undo* is rather like having your actions recorded by a video, which you can rewind to find previous states that you want to restore. However, the video keeps recording even when you are rewinding. Rewinding ordinary recording is *Undo*, and rewinding past a

previous rewinding is *Redo*! Possibly a real-time or video-player metaphor would make such an *Undo* /*Redo* policy more comprehensible.

5.5.4 Commitment points

In the taxonomy proposed in Figure 5.1, we said that for systems of class (i) (single-action *Undo*), each active command commits the previous one, so it cannot longer be undone. Instead, systems of class (ii) (backtrack/single-action) and of class (iv) (backtrack/multiple-action) have no backward commitment points.

For systems of class (i) adding of *Redo* does not modify the commitment point, which is still introduced by the first active command after the *Undo* /*Redo*.

Conversely, for systems of class (ii) and (iv) the situation is a little different. In fact, adding *Redo* creates an 'undo phase'. Actually, the user can type a sequence of active, passive or neutral actions, i.e. the ordinary editing phase, and can then start the undo phase. The last may be composed by a sequence of single-action *Undo* /*Redo* (systems of class (ii)), or by a sequence of multiple-action *Undo* /*Redo* (systems of class (iv)). Any *Undo* /*Redo* may be followed and/or preceeded by a sequence (eventualy empty) of passive/neutral actions. At this point, any entered active command breaks the undo phase and creates a branch in the commands history. From a past state there are now two 'next' states, the one previously reached, and the one resulting from the new command. Some systems directly support such a branched history [12]. However, the complexity of representing this at the user interface (as well as the cost of implementing it) is high. Rather than attempting to represent a branching history in the user interface, some systems, as Word 6 and Netscape, adopt the same approach of the systems of class (i): they commit the undo phase after any active command. That is, the pending script is chopped off and *Redo* is no longer possible. This means we can regard the dialogue as a number of phases: ordinary phases consisting of a mixture of active, passive and neutral commands and undo phases consisting of a mixture of passive, neutral, *Undo* and *Redo* commands (Figure 5.3). The transition between these phases is implicit, triggered by the first *Undo* or active action.

From the above consideration, the undo phase may be seen as a subdialogue during the human-computer interaction. Out of this subdialogue, such *Undo* phase is seen as a an *Undo* of a block of actions without *Redo*. In fact, let us suppose that we have entered this sequence of actions:

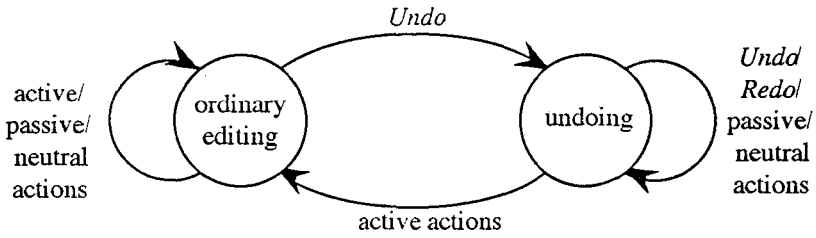

Figure 5.3: Editing and *Undo* phases represented as a state transition network.

$c_1, c_2, c_3, c_4, c_5, U(3), R(2), U(1), c_6$
where U(i) indicates that we have performed an *Undo* of the last i commands; similarly for the *Redo*.

Considering that *Undo* removes actions from the history and *Redo* add some of the undone ones, we can clump together $U(3), R(2), U(1)$ in $U(2)$. In this way, out of the undo phase, the sequence
$c_1, c_2, c_3, c_4, c_5, U(3), R(2), U(1), c_6$,
above mentioned, is seen as
$c_1, c_2, c_3, c_4, c_5, U(2), c_6$, which is equivalent to c_1, c_2, c_3, c_6.

5.6 Final analysis

Granularity dependence or independence between *Undo* / *Redo* involves different systems' behaviours in human-computer interaction. In fact, when performing *Redo*, if the previous *Undo* has been single (in this case, for granularity dependence also *Redo* is single), then it is the system which decides which state has to be reached. On the contrary, if there is no granularity dependence, it is the user that chooses which state he is interested in reaching. The fact that the user can choose how to modify the history brings, as a consequence, a different representation on the screen of the interactive recovery functionality: we have no more buttons or icons, but lists. In fact, it is difficult to represent visually something extremely linear as history.

An example of linearity of history is represented by VisEd [55], a visual editor used to formulate a query to a database of images. In it, a query is given by a sketch of the image the user is interested for retrieval. In this application there are two levels of interaction: 1) while interacting with the

visual editor, and 2) while interacting with the database. In the first case, the interface supports an explicit *Undo* whose semantics is based on the flip *Undo*. In the second case, an *Undo* mechanism is provided by a browser which allows the user to reach (and eventually modify, applying the principles of progressive querying) any query formulated during his interaction. The query history is linear: it is handled as a list and the browser moves as a pointer in the list. The visual approach is used for the interaction, while the history is linearly represented, query by query.

There are also some interactive systems which have a visual representation of the history. An example is provided by Hypercube [20], that is a visual query system which support the progressive querying technique. The history is composed by different layers, anyone representing a formulated query which, when overlapped, creates a cube; it allows to move queries, and, eventually, to modify them, dragging a layer from backward to forward. However, the hypercube technique cannot be applied to a text editor, for the high number of states which may belong to the history of a document.

As we said in the previous Chapter, what we expect when interacting with a computer, is to obtain a large effect from small actions. For this reason a very powerful *Undo*, as for systems of class (iv), is extremely useful. But we not reduce the risk in interaction by increasing the power of *Undo*! In fact, as it happens in Netscape and Word 6.0, the user has the ephemeral idea to be able to reach any state in the past, since, in different ways for different systems, he can handle the list of the *Undo* and *Redo* functions. In practice, he can move between the two contexts, *Undo* and *Redo*, until he does not reach a branching point and follows another path. In this way the old branch is deleted from the *Undo* or *Redo* lists. For this reason, although multiple-undo is more powerful that the single one since it allows the user to handle directly all the past history, it is also more dangerous. In fact, if a user cancels many actions with only one *Undo* (in Word 6 he can cancel up to 100 actions!) and then he realises that the reached state is wrong but, accidentally, makes a slip, for example touches the space-bar, then a branching point is created and all the past history is lost: with a very small action (as a key press) we can have a big damage.

At the beginning of this Chapter, we have introduced *Undo* as a special case of reachability, as it allows the user to reach a past state. But how can we informally define the *Redo*?

The answer is in Figure 5.4. Let's suppose that, starting from the initial state s_0 we have reached state s_n by performing only commands. At this

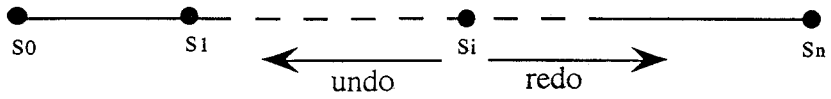

Figure 5.4: The linearity of the history: *Undo* is the past in the past, Redo is the future in the past.

point the set of the past state is $\{s_0, s_1, ..., s_{n-1}\}$. Suppose also that performing a multiple *Undo* we have reached state s_i. Now, s_i is the current state and the set of states which *Undo* allows to reach is $\{s_0, s_1, ..., s_{i-1}\}$. So *Undo* continues to be a function in the past, allowing the user to go to the left side of the past history. In the current state, we can say that *Undo* is the past of the past. Yet, the set of states which *Redo* allows to reach is $\{s_{i+1}, s_{i+2}, ..., s_n\}$. From the current state s_i *Redo* is a function into the future, allowing the user to go to the right side of the past history. So also *Redo* is a special case of reachability, allowing the user to reach a future state in the past.

Chapter 6

Formal behaviour of backtrack Undo

In this Chapter we are going to analyse and formally express the relationships between an original system without *Undo* and its augmented one, which is enriched by *Undo*. Such a formalisation will be done through the definition of conservative encapsulation. The last does not consider the kind of *Undo* (for example if flip or backtrack *Undo*), but simply captures the idea that the original system is, in some way, still 'inside' the full system with *Undo*. When we talk about a particular kind of *Undo*, the conservative encapsulation is not sufficient to capture all the aspects of its behaviour. To this aim we will introduce two equivalence relationships, a strong one and another which is weaker, and we will provide four equations, based on the above mentioned equivalences and on the monotone one, which fully describe the behaviour of flip and backtrack *Undo*. Next, we will focus only on the backtrack *Undo* and, after providing its formal definition, we will prove that all the backtrack *Undo* of the same PIE are behaviourally equivalent.

6.1 Systems with and without Undo

In the previous Chapter, when we provided an informal definition of *Undo*, we referred to it as a system function which allows the user to reach "any" or "the" previous state. But it is difficult to provide a definition of state. A state represents the situation in which the system is at a given time, and such a state may have more or less components, depending on what we consider as a system, and hence on how many components the system has.

One definition of 'state' is the one we have in the 'state' component of the ACS model (see Chapter 5). This corresponds to the state of the system if there were no *Undo*. Indicating a system without *Undo* as $\mathcal{P}$, we may call S the set of states of $\mathcal{P}$.

If we add *Undo* to the original system $\mathcal{P}$, we have a new system $\mathcal{P}^u$. In this case, in order to be able to perform *Undo*, $\mathcal{P}^u$ must store additional information, often some sort of history or record of past states. That is, the full state of the system contains more information than in S. This complete state of *all* the system, including the bits needed for *Undo*, we refer to as S^u.

In the same way in which we have considered two kinds of state, as the state of a system without *Undo* and the state of the same original system enriched by *Undo*, we can consider two kinds of action history, one related to $\mathcal{P}$ and the other to $\mathcal{P}^u$. In the case of $\mathcal{P}$, we can indicate the command history with H, similarly we can use H^u for $\mathcal{P}^u$.

What we are going to do in the next Sections, is to formalise the relationships between $\mathcal{P}$ and $\mathcal{P}^u$, expressing some formal link between H and H^u, S and S^u. In particular, we will introduce the definition of *conservative encapsulation* in order to express the idea that the original system is, in some way, still there 'inside' the full system with *Undo*.

We firstly consider the system without *Undo*, then look at the full system, and finally the relationship between the two. The model we will use is a form of the PIE model [27], using multiple levels of abstraction as in [29].

6.1.1 System without Undo

In order to study the relationship between $\mathcal{P}$ and $\mathcal{P}^u$, we need to be able to establish if two histories produce the same effect. The word 'same' implies an equivalence relationship. Since to understand the *Undo* mechanism we need a deep knowledge of the system behaviour, we require to know if two effects are the same when they look the same. For this reason, we need to use the monotone equivalence, with which we can be sure that what looks the same is really the same. If a PIE is monotone, that is the monotone equivalence holds, we can talk indifferently of effects or states. By indicating with C the set of 'ordinary' commands (i.e. not *Undo*), we can define a state update function *doit* as

$$doit : S \times C \rightarrow S$$

with an initial state s_0.

We can derive from this function two other functions: $doit^*$, obtained by iterating *doit*, and I, the *interpretation* function of the PIE model:

$$doit^* : S \times H \rightarrow S$$

where

$$doit^*(s, <>) = s$$
$$doit^*(s, h \frown c) = doit(doit^*(s, h), c)$$

This iterated version tells us the effect of a whole sequence of commands. Recall that the sequence of commands, written as H, the command history, is defined by $H = C^*$, the set of finite sequences of C.

Since $\mathcal{P}$ is monotone, we can define the interpretation function in term of the *doit*. In this case, we can simply define the interpretation function as the iterated *doit* starting from the initial state:

$$I(h) = doit^*(s_0, h)$$

We will also use a dot to represent the 'curried' version of a *doit* function:[1]

$$doit(., c) : S \rightarrow S$$

where

$$doit(., c) = \lambda s \bullet doit(s, c)$$

6.1.2 System with Undo

When we consider the system with *Undo*, as we noted, the state space increases. The set of full states we call S^u and the set of actions $A = C \cup U$, the last expresses that any user action may be an ordinary command $c \in C$ or an *Undo*. As for the system without *Undo*, we have a corresponding state update function $doit^u$ and initial state s_0^u. As with the original system we can define an iterated version $doit^{u*}$. For the same reason that applies for $\mathcal{P}$, also $\mathcal{P}^u$ is monotone; this allows us to define an interpretation function $I^u = doit^{u*}(s_0, h)$.

It is important to note that this full state will extend the original state, not in the sense that there are extra possible states (i.e. *not* $S \subset S^u$),

[1] Currying is a technique used in functional programming and lambda calculus to simplify the presentation of complex formulae. Some of the parameters of a function are fixed, giving a function with fewer parameters. In this case, we are fixing the command parameter of *doit*, yielding a function $doit(., c)$, which has one parameter, i.e. only a state.

but in the sense that each state of the full system has some component (or effectively such) that corresponds to a state of the original system. That is, there is a projection function *proj*, which, given a state of the full system, gives a corresponding state of the original system.

$$proj : S^u \rightarrow S$$

Typically, the full state contains some form of history information. For example, a particular *Undo* system might store the 'normal' state and also the command history (active script). That is, its state would be given by:

$$S^u = S \times H \qquad (example\ state)$$

The projection function would then be:

$$\forall < s, h > \in S^u \bullet proj(< s, h >) = s \qquad (example\ projection)$$

The exact way in which the original state is extended, and the nature of the projection function, will differ between *Undo* functions.

6.2 Encapsulation

When we add *Undo* to a system $\mathcal{P}$ we expect that, in some sense, the original system is still inside the augmented one, that is we expect that they behave similarly when we do not perform *Undo*. We can show it by proving different theorems, which are based on different definitions, starting from the one of *encapsulation*.

Definition 6.1 (Encapsulation) *Given a system* $\mathcal{P} = < H, S, doit, s_0 >$ *we say that its augmented system* $\mathcal{P}^u = < H^u, S^u, doit^u, s_0^u >$ *is an* encapsulation *of* $\mathcal{P}$ *if there exist two functions, proj and eff, such that:*

(i) $proj : S^u \rightarrow S$ (*C1*)

(ii) $eff : H^u \rightarrow H$ (*C2*)

(iii) $\forall h \in H^u \bullet proj(I^u(h)) = I(eff(h))$ (*C3*)

Condition (i) represents a link between the sets of states S^u and S. Condition (ii) represents a mapping, between the histories. Such a mapping, that we indicate with *eff*, corresponds to the mapping in which the ACS model determines the active script from the user history. Finally, condition

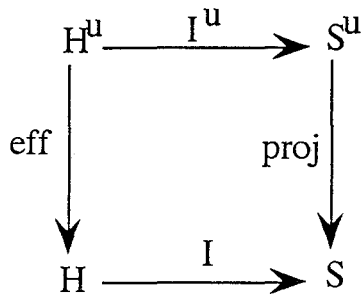

Figure 6.1: Encapsulation

(iii) says that the part of the state corresponding to the original system is just as if we had executed the effective history. Indeed, the system may actually be implemented by using the original update functions on this part of the system state. Note that, this condition says nothing about the way in which the effective history is related to the action history, merely that it and the projected part of the state 'agree'.

The conditions for an encapsulation can be summarised by the commuting diagram in Figure 6.1. The two sides of the above equation correspond to the two paths round the diagram.

6.2.1 Conservativeness of state and history

The encapsulation condition says that the original system is still in the augmented one. However, so far we have set no conditions other than that the effective history and the projection in some sense agree. We want to say more. Obviously the new commands may have arbitrary behaviour, but we expect the original commands to behave as they always did on the original part of the state. In keeping with other areas of formal specification, we regard this as a *conservativeness* property – the original system is conserved within the extended system. This is formally expressed by the following definition:

Definition 6.2 (Conservativeness of state) *Given a system $\mathcal{P}^u$ which is an encapsulation of the original system $\mathcal{P}$, we have the conservativeness of state if:*

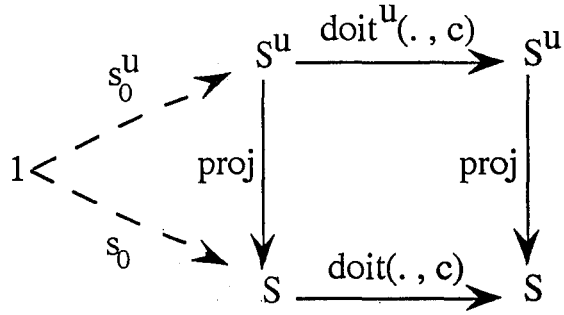

Figure 6.2: Conservativeness of state.

(i) $proj(s_0^u) = s_0$ *(C4)*

(ii) $\forall c \in C, s \in S^u \bullet proj(doit^u(s, c)) = doit(proj(s), c)$ *(C5)*

Condition (i) says that the initial state of the full system (s_0^u) corresponds (via the projection function) to the initial state of the original system (s_0); condition (ii) says that the effect of applying a command to the full state is the same as that of applying it to the projected form of the state. This condition may be captured in a commuting diagram, shown on Figure 6.2. The main part of the diagram corresponds to condition (ii), and the small triangle on the left to condition (i). The '1' refers to the set of one element and the arrows labeled 's_0^u' and 's_0' are constant mappings (from the single element of '1'). This is simply a formal trick that allows us to include this information on the diagram. Also note that the functions at the top and bottom of the diagram are the curried versions of the appropriate *doit* functions. They are for a particular command c, and strictly one can imagine a copy of this diagram corresponding to every such command.

In a similar fashion we expect the effective history to behave in a sensible fashion where ordinary commands are concerned. To express this, we introduce the definition of *conservativeness of effective history.*

Definition 6.3 (Conservativeness of effective history) *Given a system $\mathcal{P}^u$ which is an encapsulation of the original system $\mathcal{P}$, we have the conservativeness of effective history if:*

(i) $eff(<>) = <>$ *(C6)*

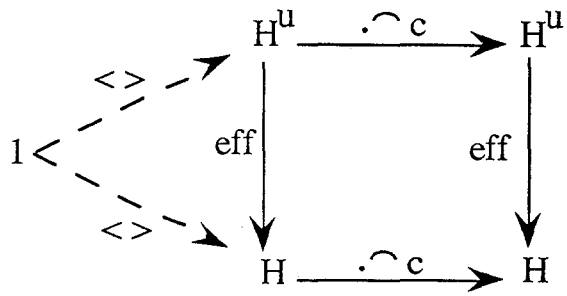

Figure 6.3: Conservativeness of effective history.

(ii) $\forall c \in C, h \in H^u \bullet eff(h \frown c) = eff(h) \frown c$ (C7)

Condition (i) means that the effective history corresponding to an empty action history should be empty, while the meaning of (ii) is that by adding an ordinary command to the action history, the same command is addedto the effective history. Also in this case, the above introduced conditions are captured in the commuting diagram, shown in Figure 6.3.

At the right-hand side of the definitions of encapsulation and conservativeness of state and history that we provided in the previous Sections, we put a label, an uppercase C followed by a number. Such labels will be used in the following as a quick reference to such conditions.

6.2.2 Conservative extension – the cube

If all the three diagrams commute, we will say that the augmented system $\mathcal{P}^u$ is a *conservative encapsulation* of the original system $\mathcal{P}$. More formally, we have the following definition:

Definition 6.4 *If the augmented PIE* $\mathcal{P}^u = < H^u, S^u, doit^u, s_0^u >$ *is an encapsulation of the PIE* $\mathcal{P} = < H, S, doit, s_0 >$, *and the conservativeness of state and history hold, then* $\mathcal{P}^u$ *is a conservative encapsulation of the original PIE* $\mathcal{P}$.

The whole set of conditions can be captured in a single commuting diagram (Figure 6.4), which we call 'the cube'. This diagram is rather complicated to read on its own, as it includes the diagrams related to encapsulations and both to the conservativeness of state and history. The front and back

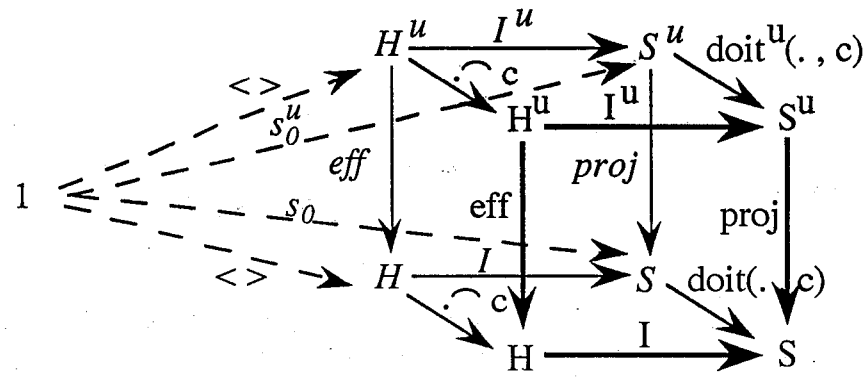

Figure 6.4: The cube.

faces of the cube are two copies of Figure 6.1, the left being Figure 6.3 and the right Figure 6.2. To make it easier to read, the captions at the back are italicsized and those at the front emboldened.

The cube has six faces: four correspond to the commuting diagrams, but that leaves out the top and bottom faces. Drawing the bottom on its own, gives the diagram in Figure 6.5. This refers only to the model of the original system, and upon examination is simply a restatement of the construction of I from *doit*. The top triangle is the initial condition that

$$I(<>) = s_0$$

and the square corresponds to the iterated case

$$I(h \frown c) = doit(I(h), c)$$

The top is similar, except that it refers to the full system.

Both the top and the bottom of the cube commute by the definitions of I and I^u. This is important, as it suggests that some faces of the cube are redundant, in the sense that they are implied by the others. In particular, when S^u represents the sets of all the reachable states, then the encapsulation and the conservativeness of the history properties imply the conservativeness of the state. The fact that S^u represents the sets of all the reachable states means that I^u is surjective, i.e. the weak reachability property holds (see

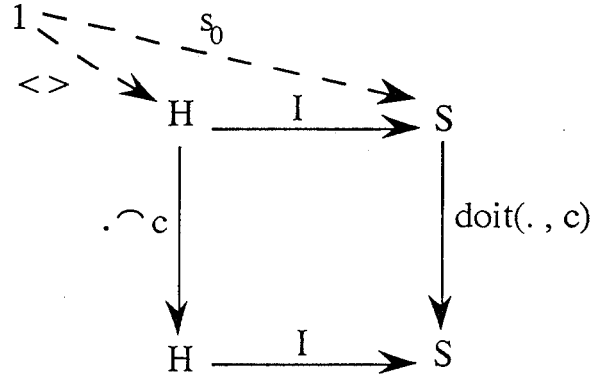

Figure 6.5: Bottom of the cube.

Chapter 1). This allows us to say that for any state s in S^u, there is a corresponding action history h from H^u which gives rise to s (i.e. $s = I^u(h)$).

It is important to consider only the reachable states, because, depending on how one formulates the state of a system, it may include so called 'garbage' states. These are states allowed by the description, but can never occur in a real system. For example, if the position of the cursor in a text editor is represented by an integer denoting the offset into the text, then it will always point to a position within the text. However, a simple definition of the state as:

$$S = Text \times Int$$

would in principle include unreachable states such as < "hello", 500 >, where the cursor points to a location outside the text. Such a state would *never* be reached during the normal use of the system and is thus 'garbage' in the formal description. We do not want, nor need to say anything about the properties of such states, they never happen and are uninteresting.

The fact that the encapsulation and the conservativeness of history imply the conservativeness of state when I^u is surjective, is proved by the following theorem:

Theorem 6.1 *Given a PIE* $\mathcal{P}^u = < H^u, S^u, doit^u, s_0^u >$ *which is an encapsulation of the PIE* $\mathcal{P} = < H, S, doit, s_0 >$, *where* S^u *is the set of reachable*

states of $\mathcal{P}^u$, and the property of conservativeness of history holds, then the property of conservativeness of state holds.

Proof: To prove this theorem, have to prove that, if both the diagrams of Figure 6.1 and Figure 6.3 commute, then also the diagram of Figure 6.2 commutes. We will consider the two parts of Figure 6.2, the left triangle and the main square, separately. In the left triangle we have to prove that the initial states agree, that is we have to prove that:

(i) $proj(s_0^u) = s_0$

while in the main square we have to prove that:

(ii) $proj(doit^u(s, c)) = doit(proj(s), c)$

We firstly prove (i). By definition of I^u, since $I^u(<>) = s_0^u$, we have

$$proj(s_0^u) = proj(I^u(<>))$$

Now, since $\mathcal{P}^u$ is an encapsulation of $\mathcal{P}$, then condition C3 holds, that is:

$$proj(I^u(<>)) = I(eff(<>))$$

For the conservativeness of history, we have that C4 holds, that is:

$$I(eff(<>)) = I(<>)$$

Finally, by definition of I, we have:

$$I(<>) = s_0$$

Clumping all these equalities, we have:

$$\begin{array}{lcll} proj(s_0^u) & = & proj(I^u(<>) & \text{defn. of } I^u \\ & = & I(eff(<>)) & \text{C3} \\ & = & I(<>) & \text{C4} \\ & = & s_0 & \text{defn. of } I \end{array}$$

□

To prove (ii), applying the weak reachability property, we can choose h_s such that:

$$s = I^u(h_s)$$

Given this definition, we have that:

$$proj(doit^u(s, c)) = proj(doit^u(I^u(h_s), c))$$

By definition of I^u, we have that:

$$proj(doit^u(I^u(h_s), c)) = proj(I^u(h_s \frown c))$$

Since $\mathcal{P}^u$ is an encapsulation of $\mathcal{P}$, the condition C3 holds, that is:

$$proj(I^u(h_s \frown c)) = I(eff(h_s \frown c))$$

Since the conservativeness of history holds, then, for condition C5 we have:

$$I(eff(h_s \frown c)) = I(eff(h_s) \frown c)$$

By definition of interpretation function I, we have:

$$I(eff(h_s) \frown c) = doit(I(eff(h_s)), c)$$

At this point, by applying condition C3 to the argument of the ***doit*** function, we have:

$$doit(I(eff(h_s)), c) = doit(proj(I(h_s), c)$$

Finally, by appying again the reachability property, we have:

$$doit(proj(I(h_s), c) = doit(proj(s), c)$$

Again, clumping all these equalities together, we have:

$$\begin{array}{rcll} proj(doit^u(s, c)) & = & proj(doit^u(I^u(h_s), c)) & \text{reach. prop.} \\ & = & proj(I^u(h_s \frown c)) & \text{defn. of } I^u \\ & = & I(eff(h_s \frown c)) & \text{C3} \\ & = & I(eff(h_s) \frown c) & \text{C5} \\ & = & doit(I(eff(h_s)), c) & \text{defn. of } I \\ & = & doit(proj(I^u(h_s)), c) & \text{C3} \\ & = & doit(proj(s), c) & \text{reach. prop.} \end{array}$$

□□

A consequence of this theorem is that, if S^u is the set af reachable states, then to verify that a particular *Undo* system is indeed a conservative encapsulation, it is sufficient to show that it satisfies the encapsulation conditions and that the effective history behaves appropriately, since the conservativeness of state is given by the above mentioned theorem. This is expressed by the following corollary:

Corollary 6.1 *If a PIE* $\mathcal{P}^u = < H^u, S^u, doit^u, s_0^u >$, *where* S^u *is the set of reachable states, is an encapsulation of the PIE* $\mathcal{P} = < H, S, doit, s_0 >$ *and the conservativeness property holds for the effective history, then* $\mathcal{P}^u$ *is a conservative encapsulation of* $\mathcal{P}$.

6.3 Algebraic properties for Undo

In the previous Chapter we introduced a classification of *Undo* systems, mainly based on the fact that *Undo* may belong to the command history or not. In the last case, *Undo* is not self-applicable (i.e. the following *Undo* does not reverse the effect of the previous one) and the *Undo* of the *Undo* may be used as a backtrack tool. With the pure backtrack *Undo* mechanism, for any application of *Undo*, the system totally forgets about the cancelled command and no form of *Redo* will be possible. The fact that, with the pure backtrack *Undo* the system forgets everything about the cancelled command, allows the user to reach "exactly" the previous state. Using the strong equivalence introduced in Chapter 1, we have that

$$c \frown Undo \sim null \qquad \text{E1}$$

When *Undo* belongs to the command history, then it is self applicable and the *Undo* of *Undo* is allowed as the *Redo* function. We refer to this class of *Undo* mechanism as flip *Undo*. With the flip *Undo*, after an *Undo* application, the user has the sensation that the reached state is the previous one. In fact, the internal component of the state has also to remember the last undone command in order to use this information if a *Redo* occurs. Since the link between states and effects is expressed by the *proj* function as $proj(s) = e$, then we can formalise the behaviour of *Undo* in flip *Undo* by introducing the projection equivalence $\sim_{proj}$. We say that two states s_1, s_2 are projection equivalent, $s_1 \sim_{proj} s_2$, if they have the same projection, that is $proj(s_1) = proj(s_2)$ -the $\sim_{proj}$ for the state corresponds to the $\equiv_I$ for the programs (see Chapter 1). In case of the flip *Undo*, we have that

$$\forall s \in S^u, c \in C \bullet proj(doit^{u*}(s, c \frown Undo)) = proj(s)$$

or, alternatively

$$c \frown Undo \sim_{proj} null \qquad \text{E2}$$

Since, if two states are strongly equivalent then they are the same, in the sense that they are equal component by component, then they will also have the same projection, i.e. the "normal" component of the state. This means that $E1 \Rightarrow E2$.

The equation E2 is not sufficient to describe flip *Undo* behaviour, since *Undo* of *Undo* may be used as the *Redo* function, reaching so "exactly" the state in which the system was before the first *Undo*. This may be expressed by the following equation:

$$Undo \frown Undo \sim null \qquad \text{E3}$$

In both the *Undo* mechanisms, backtrack and flip *Undo*, we have not yet considered what happens when *Undo* is applied at the initial state, when no command has been yet entered. Generally, in this situation *Undo* has no effect, and this is expressed in the following equation by using the monotone equivalence:

$$Undo \equiv^{\dagger} null \qquad \text{E4}$$

We now have the necessary information to formally express the behaviour of backtrack and flip *Undo*. The backtrack *Undo* may be described by the two following equations:

$$c \frown Undo \sim null \qquad \text{E1}$$
$$Undo \equiv^{\dagger} null \qquad \text{E4}$$

while the flip *Undo* behaviour can be described by the three following equations:

$$c \frown Undo \sim_{proj} null \qquad \text{E2}$$
$$Undo \frown Undo \sim null \qquad \text{E3}$$
$$Undo \equiv^{\dagger} null \qquad \text{E4}$$

At the right-hand side of the above introduced equations, we put a label, an uppercase E followed by a number. Such labels will be used in the following as a quick reference to such equations.

Notice that none of these equations capture the strongest informal definition of *Undo* that we provided in the previous Chapter. In such a definition we said that *only* and *at most* the previous state can be reached – that is, multiple *Undo* is not allowed. The formal definitions are permissive: saying what you can do, but not restrictive: saying what you cannot do. Such restrictive conditions are hard to express over the state, but can be formulated using the effective history. We can say that the effective history's lenght is never more than one, less than the longest it has ever been:

$$len(eff(h)) + 1 \geq \max_{h' < h} len(eff(h'))$$

Note that the less than or equal relation '$\leq$' is being used as a shorthand for 'is an initial subsequence of'. That is:

$$h' \leq h \iff \exists h'' \mathsf{st}. h' \frown h'' = h$$

6.4 The reflexive nature of Undo

In the previous Section, we introduced four equations in order to formally describe the behaviour of backtrack and flip *Undo*. In particular, the equations E3 and E1 characterise systems for which *Undo* is or is not self-applicable. The combination of the two equations gives a *strong Undo property* which has been called *thoroughness* [88]. However, it turns out to be effectively inconsistent. Yang [88] proves that the two common forms of *Undo* system -backtrack and flip *Undo* - do not satisfy this strong *Undo* property. In fact, it is shown in [25] that *no Undo* system can satisfy this property except for those where the underlying system has at most two states, as the diagram of Figure 6.6 shows.

The top and bottom routes round this diagram consider two different potential interactions from an arbitrary state of the system s_0. Following the upper interaction path in Figure 6.6, the user can reach s_a by executing a, while following the other route, the user can reach s_b by executing b. By performing the *Undo* in the reached state, both the routes go back to the original state s_0, as both $a \frown Undo$ and $b \frown Undo$ are equivalent to the null command (doing nothing). Finally, consider what happens if *Undo* is issued from the state s_0. Considering the upper interaction path of Figure 6.6, the following performed *Undo* in the state s_0 should lead to s_a, while, from the lower interaction path, one would conclude that executing *Undo* in the state s_0 should lead to s_b. Which is right?

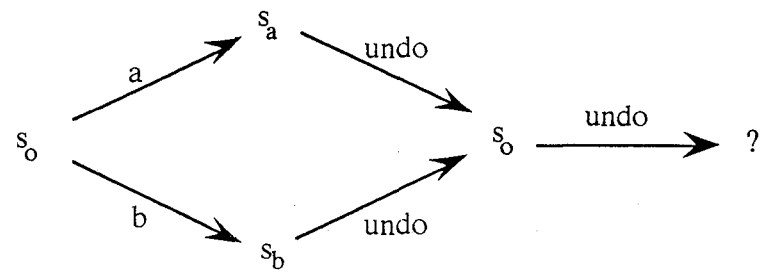

Figure 6.6: Undo of undo?

Well, if the strong *Undo* property really holds, then both must be right. That is, $s_a = s_b$. But as a and b were arbitrary commands, this means the effect of *any* command in the state s_0 is the same. Since a and b are arbitrary (one of them could also be *Undo*), we see that the system can have at most two states, with all commands (and *Undo*) simply toggling between them. That is, the strong *Undo* property is impossible to satisfy for any realistic system, which usually has more than two states.

In other words, although *Undo* is reflexive in the sense that it looks in on the interaction history of the system, it cannot be entirely reflexive, treating itself on a par with other commands.

6.5 Behavioural equivalence of backtrack undo

In the previous Sections we introduced the relationships between a PIE without *Undo* and its augmented system, the one enriched by *Undo*. Now we focus our attention only on a given class of *Undo* systems: the backtrack *Undo*.

Definition 6.5 (Backtrack undo) *A PIE* $\mathcal{P}^b = < H^b, S^b, doit^b, s_0^b >$ *is backtrack Undo of the PIE* $\mathcal{P} = < H, S, doit, s_0 >$ *if:*

(i) equations E1 and E4 hold;

(ii) $\mathcal{P}^b$ *is a conservative encapsulation of* $\mathcal{P}$ *.*

The meaning of equation E1 is that if we perform *Undo* in the initial state as a first command, we have no effect, that is

$doit^b(s_0^b, Undo) = s_0^b$. Since the system is monotone, we can express the interpretation function I in terms of the *doit* function, so that equation E1 may also be expressed as
$I(Undo) = doit^b(s_0^b, Undo) = s_0^b$.

Moreover, the meaning of equation E2 is that if we perform an *Undo* after a command, then we reach exactly the state in which the system was before *Undo* was performed. Formally, this is expressed by
$doit^b(doit^b(s_h^b, c), Undo) = s_h^b$.

Two augmented systems of the same original PIE $\mathcal{P}$ may have different states, but may be identical when viewed in terms of the state of the original system (using projection). In this case, we say that they are behaviourally equivalent, that is:

Definition 6.6 (Behavioural Equivalence) *If $\mathcal{P}^1$ and $\mathcal{P}^2$ are encapsulation of $\mathcal{P}$ with the same augmented command set C^b and command history H^b, we say that $\mathcal{P}^1$ and $\mathcal{P}^2$ are* behaviourally equivalent *if*
$\forall h \in H^b \bullet proj^1(I^1(h)) = proj^2(I^2(h))$

Another way to look at this definition is to imagine that the display we see of $\mathcal{P}^1$and $\mathcal{P}^2$is via their respective projections. If this is all we can see, then they behave the same.

Given a PIE $\mathcal{P}$, all its augmented system $\mathcal{P}^b$ are behavioural equivalent, i.e. any conservative encapsulation of $\mathcal{P}$, satisfying E1 and E4, are behaviourally equivalent to the pure backtrack *Undo*. This is proved in the following theorem:

Theorem 6.2 (Behavioural equivalence of backtrack undo) *All the PIEs which are backtrack Undo of the same PIE $\mathcal{P}$ are behaviourally equivalent.*

To prove this theorem, we need the following lemma:

Lemma 6.1 *Given a PIE $\mathcal{P}^b$ which is a backtrack Undo of the PIE $\mathcal{P}$, if we do a substitution of the function eff with a special eff, eff_{sp} so defined*
$eff_{sp} : H^b \rightarrow H$

$$\begin{array}{lcl} eff_{sp}(<>) & = & <> \\ eff_{sp}(h \frown c) & = & eff_{sp}(h) \frown c \\ eff_{sp}(\text{Undo}) & = & <> \\ eff_{sp}(h \frown \text{Undo}) & = & chop_1(eff_{sp}(h)) \end{array}$$

where

$$\begin{array}{lcl} chop_1(<>) & = & <> \\ chop_1(h \frown c) & = & h \\ chop_{n+1}(h) & = & chop_1(chop_n(h)) \end{array}$$

then $\mathcal{P}^b$ is still a backtrack Undo of $\mathcal{P}$ with respect to $proj^b$ and the eff_{sp} functions.

In order to prove this lemma, we need to exploit two properties of eff_{sp}:

Proposition 6.1 *Given the eff_{sp} function so defined*

$eff_{sp} : H^b \rightarrow H$

$$\begin{array}{lcl} eff_{sp}(<>) & = & <> \\ eff_{sp}(h \frown c) & = & eff_{sp}(h) \frown c \\ eff_{sp}(Undo) & = & <> \\ eff_{sp}(h \frown Undo) & = & chop_1(eff_{sp}(h)) \end{array}$$

where

$$\begin{array}{lcl} chop_1(<>) & = & <> \\ chop_1(h \frown c) & = & h \\ chop_{n+1}(h) & = & chop_1(chop_n(h)) \end{array}$$

then, the two following properties hold :

(i) $eff_{sp}(Undo^n) =<>$ (*P1*)

(ii) $eff_{sp}(p \frown c \frown Undo \frown q) = eff_{sp}(p \frown q)$ (*P2*)

where $Undo^n$ indicates the application of n successive Undo.

Proof (Proposition 6.1): the proof of the above introduced properties is done by induction. We start with property (i).

(i_a) basic step, $n = 0$

If $n = 0$, then $Undo^n =<>$, that is $eff_{sp}(Undo^n) = eff_{sp}(<>)$.

By definition of eff_{sp}, we have that $eff_{sp}(<>) =<>$.

□

(i_b) general case

Assume that $eff_{sp}(Undo^n) =<>$ holds by inductive hypothesis, then prove that $eff_{sp}(Undo^{n+1}) =<>$ holds.

Since $Undo^{n+1} = Undo(Undo^n)$, then

$$eff_{sp}(Undo^{n+1}) = eff_{sp}(Undo(Undo^n)).$$

By definition of eff_{sp} we have that

$$eff_{sp}(Undo(Undo^n)) = chop_1(eff_{sp}(Undo^n)).$$

By applying inductive hypothesis on the argument of the $chop_1$, for which $eff_{sp}(Undo^n) =<>$, we have that

$$chop_1(eff_{sp}(Undo^n)) = chop_1(<>)$$

and by definition of $chop_1$, we have

$$chop_1(<>) =<>.$$

Clumping all these equalities together, we have:

$$\begin{array}{lcll} eff_{sp}(Undo^{n+1}) & = & eff_{sp}(Undo(Undo^n)) & def.\ of\ Undo\ seq. \\ & = & chop_1(eff_{sp}(Undo^n)) & def.\ of\ eff_{sp} \\ & = & chop_1(<>) & induc.\ hypothesis \\ & = & <> & def.\ of\ chop_1 \end{array}$$

□

Now we can prove the property (ii). This is done by structural induction on q.

(ii_a) basic step, $q =<>$

We have to prove that $eff_{sp}(p \frown c \frown Undo) = eff_{sp}(p)$

$$\begin{array}{lcll} eff_{sp}(p \frown c \frown undo) & = & chop_1(eff_{sp}(p \frown c)) & def.\ of\ chop_1 \\ & = & chop_1(eff_{sp}(p) \frown c) & def.\ of\ eff_{sp} \\ & = & eff_{sp}(p) & def.\ of\ chop_1 \end{array}$$

(ii_b) general case

We have to prove that $eff_{sp}(p \frown c \frown Undo \frown q) = eff_{sp}(p \frown q)$.

In this case, q may be a history h followed by a command or by an *Undo*

$$q = \begin{cases} h \frown c_1 & (ii_{b1}) \\ h \frown Undo & (ii_{b2}) \end{cases}$$

Assume that the inductive hypothesis is valid until h; prove it for $h+1$. The one more may be an ordinary command c or an *Undo*. Consider before the case (ii_{b1})

Left

By definition of eff_{sp} we have that:

$eff_{sp}(p \frown c \frown Undo \frown h \frown c_1) = eff_{sp}(p \frown c \frown Undo \frown h) \frown c_1$.

By applying inductive hypothesis on $(p \frown c \frown Undo \frown h)$, we have that

$$eff_{sp}(p \frown c \frown Undo \frown h) \frown c_1 = eff_{sp}(p \frown h) \frown c_1$$

Right

By definition of eff_{sp}

$$eff_{sp}(p \frown h \frown c_1) = eff_{sp}(p \frown h) \frown c_1$$

so the left-hand side and right-hand side are equal.

$\square(ii_{b1})$

Finally, consider the case (ii_{b2}), in which $q = h \frown Undo$.

Left

By definition of $chop_1$, we have

$eff_{sp}(p \frown c \frown Undo \frown h \frown Undo) = chop_1(eff_{sp}(p \frown c \frown Undo \frown h))$

Applying structural induction on $(p \frown c \frown Undo \frown h)$, we have

$$chop_1(eff_{sp}(p \frown c \frown Undo \frown h)) = chop_1(eff_{sp}(p \frown h))$$

Right

By definition of $chop_1$, we have

$$eff_{sp}(p \frown h \frown Undo) = chop_1(eff_{sp}(p \frown h))$$

so the left-hand side and right-hand side are equal.

$\square\square$

Before starting the proof of Lemma 6.1, we need to introduce (and to prove!) another property:

Proposition 6.2 *Given a PIE* $\mathcal{P}^b = < H^b, S^b, doit^b, s_0^b >$ *which is a backtrak* Undo *of the PIE* $\mathcal{P} = < H, S, doit, s_0 >$, *then the following property holds:*

$$I^b(Undo^n) = s_0^b \qquad (P3)$$

Proof (Proposition 6.2): The proof of this property is also done by induction. Since $\mathcal{P}^b$ is backtrack *Undo* of $\mathcal{P}$, the equality E4 holds, that is $I(Undo) = s_0^b$.

(i) Basic step, $n = 1$.

By applying equation E4, we have:

$$I^b(Undo^n) = I^b(Undo) = s_0^b.$$

(ii) general case

Assume the inductive hypothesis holds for n, prove it for $n+1$.

By definition of *Undo* we have that

$$I^b(Undo^{n+1}) = I^b(Undo^n \frown Undo)$$

which, by definition of I^b, is

$$= doit^b(I^b(Undo^n), Undo)$$

and the last, being $I^b(Undo^n)$ equal to s_0^b by inductive hypothesis, is equal to

$$= doit^b(s_0^b, Undo)$$

which is equal to s_0^b by definition of $doit^b$.

□□

Now, we can start the proof of Lemma 6.1.

Proof (Lemma 6.1): To prove that the PIE $\mathcal{P}^b$ with the special eff_{sp} function is a backtrack *Undo* of $\mathcal{P}$, we have to prove that:

(1) equations E1 and E4 hold,

(2) $\mathcal{P}^b$ is a conservative encapsulation of $\mathcal{P}$.

The function eff_{sp} satisfyes E4 by definition and E1 by property P2, so (1) is done. We have to prove (2). To prove that $\mathcal{P}^b$ is a conservative encapsulation of $\mathcal{P}$, we need to prove that

(i) $\mathcal{P}^b$ is an encapsulation of $\mathcal{P}$;

(ii) conservativeness of history holds;

(iii) conservativeness of state holds.

Alternatively, by applying Corollary 6.1, only points (i) and (ii) would be necessary, since (iii) is a consequence of (i) and (ii). Really, since we have modified only eff_{sp} function and not $proj^b$, and since the original $\mathcal{P}^b$ was a backtrack *Undo* of $\mathcal{P}$, then (iii) still holds. The point (ii) is simply given by definition of eff_{sp}. We must prove (i). To prove that $\mathcal{P}^b$ is an encapsulation of $\mathcal{P}$, means that the diagram of Figure 6.7 commutes. This means that we have to prove that

$$\forall h \in H^b \bullet I(eff_{sp}(h)) = proj^b(I^b(h)).$$

The proof is provided by induction.We have to consider three cases:

(a) basic step, $h = <>$;

(b) $h = h' \frown c$;

(c) $h = h' \frown Undo$

Really, for case (c), we have to consider two kinds of history. In fact, the last may be done by a sequence of n *Undo*, or by a sequence in which at least one element is an ordinary command. These two further conditions are expressed respectively in

(c_1) $h = Undo^n$;

(c_2) $h = h_k \frown c \frown Undo^{n-k-1}$;

where h_k is a sequence of k elements of H^b.

(a) basic case, $h = <>$.

In this case we have to prove that

$$proj^b(I^b(<>)) = I^b(eff_{sp}(<>))$$

Left

By definition of I^b we have that

$$proj^b(I^b(<>)) = proj^b(s_0^b).$$

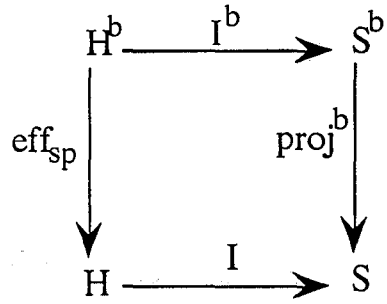

Figure 6.7: Encapsulation with the eff_{sp} function

Since there is conservativeness of state, then $proj^b(s_0^b) = s_0^b$.
Right

By definition of eff_{sp} we have

$$I^b(eff_{sp}(<>)) = I^b(<>)$$

which is equal to s_0^b by definition of I^b.

$\Box(a)$

(b) Assume that $\forall h \in II^b \bullet proj^b(I^b(h)) = I(eff_{sp}(h))$,
then we have to prove that
$\forall h \in H^b, c \in C \bullet proj^b(I^b(h \frown c)) = I(eff_{sp}(h \frown c))$.
Left

By definition of I^b we have that

$$proj^b(I^b(h \frown c)) = proj^b(doit^b(I^b(h), c))$$

By applying conservativeness of state, we have:

$$proj^b(doit^b(I^b(h), c)) = doit(proj^b(I^b(h), c))$$

By applying inductive hypothesis on h, we have:

$$doit(proj^b(I^b(h), c)) = doit(I(eff_{sp}(h), c))$$

Finally, by using the relationship between *doit* and I, we have:

$$doit(I(eff_{sp}(h), c)) = I(eff_{sp}(h) \frown c)$$

Right

By definition of eff_{sp} we have that

$$I(eff_{sp}(h \frown c)) = I(eff_{sp}(h) \frown c)$$

so the left-hand side and right-hand side are equal.

$\square(b)$

(c_1) In this case $h = Undo^n$.

Assume that $proj^b(I^b(Undo^n)) = I(eff_{sp}(Undo^n))$,
then we have to prove that
$proj^b(I^b(Undo^n \frown Undo)) = I(eff_{sp}(Undo^n \frown Undo))$.

Left

For P3 in Proposition 6.2 we have that $I^b(Undo^n) = s_0^b$, so

$$proj^b(I^b(Undo^n \frown Undo)) = proj^b(I^b(Undo^{n+1})) = proj^b(s_0^b)$$

By applying conservativeness of state, we have $proj^b(s_0^b) = s_0$.

Right

By applying P1 of Proposition 6.1, for which $eff_{sp}(Undo^n) =<>$, we have that

$$I(eff_{sp}(Undo^{n+1})) = I(<>)$$

and, by definition of I, we have $I(<>) = s_0$.
So, the left-hand side and right-hand side are equal.

$\square(c_1)$

(c_2) In this case $h = h_k \frown c \frown Undo^{n-k-1}$.

Assume that
$proj^b(I^b(h_k \frown c \frown Undo^{n-k-1})) = I(eff_{sp}(h_k \frown c \frown Undo^{n-k-1}))$,
then we have to prove that
$proj^b(I^b(h_k \frown c \frown Undo^{n-k})) = I(eff_{sp}(h_k \frown c \frown Undo^{n-k}))$.

Left

By definition of $Undo$, we have
$proj^b(I^b(h_k \frown c \frown Undo^{n-k})) = proj^b(I^b(h_k \frown c \frown Undo \frown Undo^{n-k-1}))$
which, by applying equation E1, becomes
$= proj^b(I^b(h_k \frown Undo^{n-k-1}))$

Now, being the sequence $h_k \frown Undo^{n-k-1}$ shorter than $n+1$, then, by applying structural induction we have:

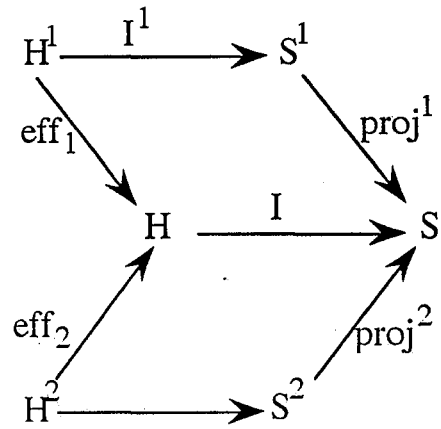

Figure 6.8: $\mathcal{P}^1$and $\mathcal{P}^2$are encapsulation of the same PIE $\mathcal{P}$.

$$proj^b(I^b(h_k \frown Undo^{n-k-1})) = I(eff_{sp}(h_k \frown Undo^{n-k-1})).$$

Right

Since eff_{sp} satisfies equation E1 (P2 in Proposition 6.2), we have:

$$I(eff_{sp}(h_k \frown c \frown Undo^{n-k})) = I(eff_{sp}(h_k \frown Undo^{n-k-1}))$$

so the left-hand side and right-hand side are equal.

□□

Finally, we can now provide the proof of Theorem 6.2:

Proof (Theorem 6.2, Behavioural equivalence of backtrack *Undo*): Let us consider two PIEs, $\mathcal{P}^1 = < H_1, S_1, doit^1, s_0^1 >$ and $\mathcal{P}^2 = < H_2, S_2, doit^2, s_0^2 >$ which are backtrack *Undo* of the same PIE $\mathcal{P} = < H, S, doit, s_0 >$. We have to show that they are behaviourally equivalent, that is the diagram of Figure 6.8 commutes.

Since $\mathcal{P}^1$and $\mathcal{P}^2$are bactrack *Undo* of the same PIE $\mathcal{P}$, then $H^1 = H^2 = H^b = H \cup Undo$. For this reason, the diagram of Figure 6.8 may be redrawn as in Figure 6.9. To prove the commutativity of this diagram, we have to prove that $\forall h \in H^b, proj^1(I^1(h)) = proj^2(I^2(h))$. By applying Lemma 6.1, if we make a substitution of eff_1 and eff_2 with eff_{sp}, then the

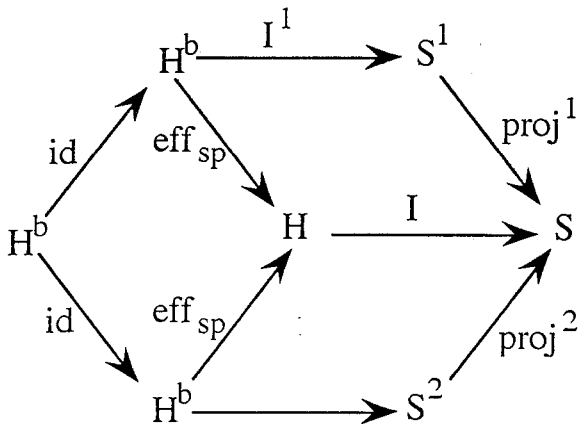

Figure 6.9: Another way to represent that $\mathcal{P}^1$ and $\mathcal{P}^2$ are encapsulation of the same PIE $\mathcal{P}$.

upper and lower parallelograms of Figure 6.9 still commute. So, by applying the commutativity of the upper parallelogram of Figure 6.8 (due to the fact that $\mathcal{P}^1$ is an encapsulation of $\mathcal{P}$) we have:

$$\forall h \in H^b, proj^1(I^1(h)) = I(\epsilon ff_1(h)).$$

By applying Lemma 6.1, we have

$$\forall h \in H^b, I(\epsilon ff_1(h)) = I(\epsilon ff_{sp}(h)).$$

In the same way we have

$$\forall h \in H^b, proj^2(I^1(h)) = I(\epsilon ff_2(h)) = I(\epsilon ff_{sp}(h))$$

□□

Theorem 6.2 says that, even if we have different implementations of the backtrack *Undo* of a given system, they are all equivalent. Such an equivalence is in terms of the effective history, and we cannot say anything about the relationships between the sets of states of the involved PIEs. The meaning of this theorem is that, from the user point of view, all the backtrack *Undo* of the same system have the same behaviour.

Chapter 7

Upper and lower bound of backtrack Undo

In the previous chapter we proved that all the backtrack *Undo* of the same original PIE $\mathcal{P}$, even if they have different implementations, are behaviourally equivalent, that is from the user's point of view they have the same behaviour. In proving such equivalence, we established a relationship between backtrack *Undo* PIEs, a relationship based on the effective history. What can we say on the sets of states? Is it possible to find the "biggest" and the "smallest" set of states? We cannot say precisely what there will be in any set of states, but for any backtrack *Undo* $\mathcal{P}^b$ we can establish a kind of partial order, finding a lower and upper bound. We will call such lower and upper bound $\mathcal{P}^{min}$ and $\mathcal{P}^{max}$ respectively. In this Chapter we are going to prove that for any $\mathcal{P}^b$ there exists a homomorphism from the set of states of $\mathcal{P}^{max}$ and the one of $\mathcal{P}^b$, and similarly that for any $\mathcal{P}^b$ there exists a homomorphism from the set of states of $\mathcal{P}^b$ to the one of $\mathcal{P}^{min}$. At the end of the chapter, such homomorphisms will be used to provide a categorial representation of backtrack *Undo*. In fact,we will prove that the class of all the backtrack *Undo* of the same original PIE $\mathcal{P}$ is a category and that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the initial and terminal element of such a category.

7.1 The maximal backtrack Undo

In this Section, we are going to introduce a backtrack *Undo* with the characteristic that the set of state is as "big" as possible (in the sense of the reachable states). We will refer to such a PIE as $\mathcal{P}^{max}$, the maximal back-

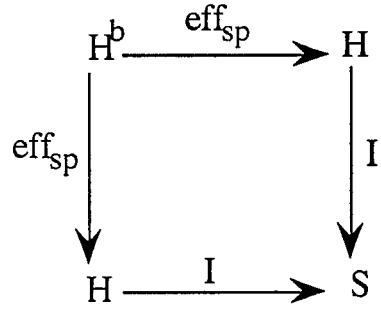

Figure 7.1: $\mathcal{P}^{max}$ is an encapsulation of $\mathcal{P}$.

track *Undo*. The word "maximal" is due to the fact that, as we will prove at the end of this chapter, $\mathcal{P}^{max}$ is maximal in the class of the backtrack *Undo* of the same original PIE $\mathcal{P}$. We can define $\mathcal{P}^{max}$ as follows:

$$\begin{array}{lcl} H^{max} & = & H^b \\ S^{max} & = & H \\ I^{max} & = & eff_{sp} \\ eff_{max} & = & eff_{sp} \\ proj^{max} & = & I(h) \end{array}$$

Such a PIE is a backtrack *Undo* of $\mathcal{P}$, and this is proved in the following Lemma:

Lemma 7.1 *$\mathcal{P}^{max}$ is backtrack Undo of $\mathcal{P}$.*

Proof: To prove this Lemma, we have to show that :

(i) $\mathcal{P}^{max}$ is a conservative encapsulation of $\mathcal{P}$;

(ii) equations E1 and E4 are satisfied.

By applying Theorem 6.1, in order to prove that $\mathcal{P}^{max}$ is a conservative encapsulation of $\mathcal{P}$, we need to prove that it is an encapsulation of $\mathcal{P}$ and the conservativeness of history holds. This means that we have to show that both the diagrams of Figure 7.1 and Figure 7.2 commute. The commutativity of diagram in Figure 7.1 is obvious, since the upper and lower path, starting from H^b to S are equal, while the commutativity of the diagram in Figure 7.2 is given by the definition of eff_{sp}.

Consider the $doit^{max}$ function and initial state s_0^{max} defined as follows:

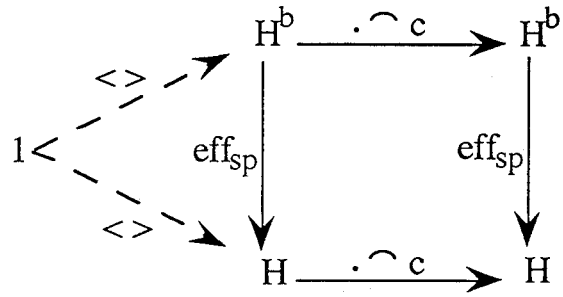

Figure 7.2: Conservativeness of history for $\mathcal{P}^{max}$.

$doit^{max} : H \times A \to H$

$doit^{max}(<>, Undo)$	$=<>$	(1)
$doit^{max}(h, c)$	$= h \frown c$	(2)
$doit^{max}(h, Undo)$	$= chop_1(h)$	(3)
s_0^{max}	$<>$	(4)

If such a $doit^{max}$ is the state update function corresponding to I^{max}, then, in order to verify that the equations E1 and E4 hold, we have to verify that

(a) $doit^{max}(s_0^{max}, Undo) = s_0^{max}$,

(b) $doit^{max}(doit^{max}(s_h^{max}, c), Undo) = s_h$

Clearly, both (a) and (b) hold, but we have to prove that $doit^{max}$ corresponds to I^{max} (and so also show that is $\mathcal{P}^{max}$ is monotone). At this aim, we need to prove that:

(i) $I^{max}(<>) = s_0^{max}$;

(ii) $I^{max}(h \frown x) = doit^{max}(I^{max}(h), x)$

where $x \in A$, that is it may be an ordinary command or an *Undo*.

We will prove before (i):

$I^{max}(<>) = eff_{sp}(<>)$	def. of I^{max}
$=<>$	def. of eff_{sp}
$= s_0^{max}$	def. of s_0^{max}

$\square(i)$

In order to prove (ii) we have to consider two cases, (ii_1) in which x is an ordinary command $c \in C$, and (ii_2) in which x is an *Undo*.

(ii$_1$) $x = c$

Left

$$\begin{aligned} I^{max}(h \frown c) &= eff_{sp}(h \frown c) && \text{def. of } I^{max} \\ &= eff_{sp}(h) \frown c && \text{def. of } eff_{sp} \end{aligned}$$

Right

$$\begin{aligned} doit^{max}(I^{max}(h), c) &= doit^{max}(eff_{sp}(h), c) && \text{def. of } I^{max} \\ &= eff_{sp}(h) \frown c && \text{def. of } doit^{max} \end{aligned}$$

So, left-hand side and right-hand side are equal.

$\square(ii_1)$

(ii$_2$) $x = Undo$

Left

$$\begin{aligned} I^{max}(h \frown Undo) &= eff_{sp}(h \frown Undo) && \text{def. of } I^{max} \\ &= chop_1(eff_{sp}(h)) && \text{def. of } doit^{max} \end{aligned}$$

Right

$$\begin{aligned} doit^{max}(I^{max}(h), Undo) &= doit^{max}(eff_{sp}(h), Undo) && \text{def. of } I^{max} \\ &= chop_1(eff_{sp}(h)) && \text{def. of } doit^{max} \end{aligned}$$

So, left-hand side and right-hand side are equal.

$\square\square$

7.2 The minimal backtrack Undo

As $\mathcal{P}^{max}$ represents the maximal backtrack *Undo*, similarly we can find the minimal backtrack *Undo*. Such a backtrack *Undo* should have the set of states as "small" as possible. We will refer to such a PIE as $\mathcal{P}^{min}$. Its characteristic is that its state is a sequence of states (of the original system) with the peculiarity that instances of the initial state are omitted from the beginning of the sequence. So, if the user in the initial state performs some actions which does not change the state, then there is no trace of such actions, since, by perfoming *Undo*, we will not change state. We can define $\mathcal{P}^{min}$ as follows:

$$
\begin{array}{lll}
H^{min} = H^b & & \\
S^{min} = S^* & & \\
s_0^{min} = <> & & \\
doit^{min} : S^{min} \times A \rightarrow S^{min} & & \\
\quad doit^{min}(<>, c) & = < doit(s_0, c) > & doit(s_0, c) \neq s_0 \\
\quad doit^{min}(<>, c) & = <> & doit(s_0, c) = s_0 \\
\quad doit^{min}(h_s, c) & = h_s \frown (doit(last(h_s), c) & h_s \neq <> \\
\quad doit^{min}(h_s, Undo) & = chop1(h_s) & \\
eff_{min} = eff_{sp} & & \\
proj^{min} : S^{min} \rightarrow S & & \\
\quad proj^{min}(<>) & = s_0 & \\
\quad proj^{min}(h_s) & = last(h_s) & h_s \neq <>
\end{array}
$$

The interpretation function I^{min} is not explicitely defined above, as it is obtained by cinstruction from $doit^{min}$ using the standard construction as introduced at the end of Section 1.7.2. As for $\mathcal{P}^{max}$, also $\mathcal{P}^{min}$ is backtrack *Undo* of $\mathcal{P}$, and this is proved by the following Lemma:

Lemma 7.2 *$\mathcal{P}^{min}$ is a backtrack* Undo *of $\mathcal{P}$.*

Proof: To prove that $\mathcal{P}^{min}$ is backtrack *Undo* of $\mathcal{P}$, we need to show that:

(i) equations E1 and E4 are satisfied;

(ii) $\mathcal{P}^{min}$ is a conservative encapsulation of $\mathcal{P}$.

To prove (i), we have to show that

(a) $doit^{min}(s_0^{min}, Undo) = s_0^{min}$;

(b) $doit^{min}(doit^{min}(h_s, c), Undo) = h_s$.

(a)

$$\begin{aligned} doit^{min}(s_0^{min}, Undo) &= doit^{min}(<>, Undo) && \text{def. of } s_0^{min} \\ &= chop_1(<>) && \text{def. of } doit^{min} \\ &= <> && \text{def. of } chop_1 \\ &= s_0^{min} && \text{def. of } s_0^{min} \end{aligned}$$

$\square(a)$

(b)

$$\begin{aligned} doit^{min}(doit^{min}(h_s, c), Undo) &= doit^{min}(h_s \frown doit(last(h_s), c), Undo) && \text{def. of } doit^{min} \\ &= chop_1(h_s \frown doit(last(h_s), c)) && \text{def. of } chop_1 \\ &= h_s && \text{def. of } chop_1 \end{aligned}$$

$\square(b)$

Now, to prove that $\mathcal{P}^{min}$ is a conservative encapsulation of $\mathcal{P}$, we need to prove that $\mathcal{P}^{min}$ is an encapsulation of $\mathcal{P}$, there is conservativeness of history and then we can apply Theorem 1. Since $H^{min} = H^b = H^{max}$ and $eff_{min} = eff_{sp} = eff_{max}$, then the diagram of the conservativeness of history for $\mathcal{P}^{min}$ is the same as in Figure 7.2, which commutes by definition of eff_{sp}. So we have only to prove that $\mathcal{P}^{min}$ is an encapsulation of $\mathcal{P}$, that is the diagram of Figure 7.3 commutes.

Such diagram commutes if

$$\forall h \in H^b \bullet I(eff_{sp}(h)) = proj^{min}(I^{min}(h)).$$

The proof is done by structural induction on h.

We have to consider the the following cases:

(1) base case $h = <>$;

(2) general case $h = h \frown c_u, c_u = c$;

(3) general case $h = h \frown c_u, c_u = Undo$.

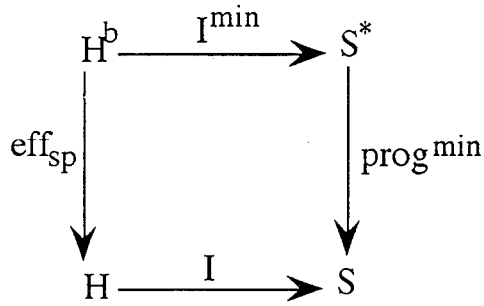

Figure 7.3: $\mathcal{P}^{min}$ is an encapsulation of $\mathcal{P}$.

(1) base case, $h = <>$

Left

$$
\begin{array}{lll}
I(eff_{sp}(<>)) & = I(<>) & \text{def. of } eff_{sp} \\
 & = s_0 & \text{def. of } I
\end{array}
$$

Right

$$
\begin{array}{lll}
proj^{min}(I^{min}(<>)) & = proj^{min}(s_0^{min}) & \text{def. of } I^{min} \\
 & = proj^{min}(<>) & \text{def. of } s_0^{min} \\
 & = s_0 & \text{def. of } proj^{min}
\end{array}
$$

□(1)

(2) General case, $h = h \frown c_u, c_u = c$.
Assume that $I(eff_{sp}(h)) = proj^{min}(I^{min}(h))$, then prove that
$I(eff_{sp}(h \frown c)) = proj^{min}(I^{min}(h \frown c))$

Left

$$
\begin{array}{lll}
I(eff_{sp}(h \frown c)) & = I(eff_{sp}(h) \frown c) & \text{def. of } eff_{sp} \\
 & = doit(I(eff_{sp}(h)), c) & \text{def. of } I \\
 & = doit(proj^{min}(I^{min}(h)), c) & \text{induc. hyp. on } I(eff_{sp}(h))
\end{array}
$$

Right

By definition of I^{min} we have that:

$$proj^{min}(I^{min}(h \frown c)) = proj^{min}(doit^{min}(I^{min}(h), c))$$

At this point, the argument of $doit^{min}$ has h as length, so we can apply structural induction. This means that, for that argument, we have a conservative encapsulation, so we can apply condition C5 (conservativeness of state) on $proj^{min}$ and $doit^{min}$, obtaining:

$$proj^{min}(doit^{min}(I^{min}(h), c)) = doit(proj^{min}(I^{min}(h)), c)$$

□(2)

(3) General case $h = h \frown c_u, c_u = Undo$.
We have two differentiate two subcases, depending on the nature of the history:

(3a) $h = Undo^n$

(3b) $h = h_k \frown c \frown Undo^{n-k-1}$

(3a)

Left

By definition of eff_{sp}, we have that

$$I(eff_{sp}(Undo^{n+1})) = I(<>)$$

that is equal to s_0 by definition of I.

Right

For property P3 of Proposition 6.2, for which $I^{min}(Undo^n) = s_0^{min}$, we have that

$$proj^{min}(I^{min}(Undo^{n+1})) = proj^{min}(s_0^{min})$$

which, by definition of $proj^{min}$, is equal to s_0. So, left-hand side and right-hand side are equal.

□(3a)

(3b) $h = h_k \frown c \frown Undo^{n-k-1}$

Left

By definition of *Undo*, we have that

$$I(eff_{sp}(h_k \frown c \frown Undo^{n-k-1} \frown Undo))$$
$$= I(eff_{sp}(h_k \frown c \frown Undo \frown Undo^{n-k-1}))$$

which, since eff_{sp} satisfies equation E1, becomes:

$$= I(eff_{sp}(h_k \frown Undo^{n-k-1}))$$

The argument of the eff_{sp} is shorter than h, so, by applying structural induction we have that

$$I(eff_{sp}(h_k \frown Undo^{n-k-1})) = proj^{min}(I^{min}(h_k \frown Undo^{n-k-1})).$$

Right

By definition of *Undo*, we have that

$$proj^{min}(I^{min}(h_k \frown c \frown Undo^{n-k-1} \frown Undo))$$
$$= proj^{min}(I^{min}(h_k \frown c \frown Undo \frown Undo^{n-k-1}))$$

Since E1 holds, we have that

$$proj^{min}(I^{min}(h_k \frown c \frown Undo \frown Undo^{n-k-1}))$$
$$= proj^{min}(h_k \frown Undo^{n-k-1})$$

So, left-hand side and right-hand side are equal.

□□

7.3 Existence of a homomorphism for P^{max}

In the previous chapter, we proved that all the backtrack *Undo* of the same PIE are behaviourally equivalent, in the sense of the effective history. But we had not yet the suitable information in order to tell something also on the sets of states of the considered backtrack *Undo*. The following theorem express a relationship between the set of states of $\mathcal{P}^{max}$ and the one of a general backtrack *Undo* $\mathcal{P}^b$, both of them backtrack *Undo* of the same

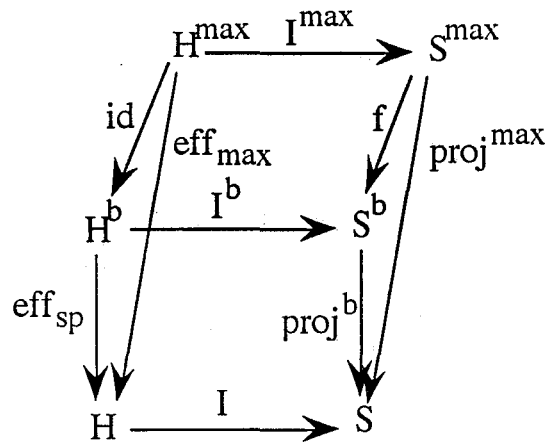

Figure 7.4: Encapsulation of the maximal PIE and of a backtrack undo of the same original PIE P.

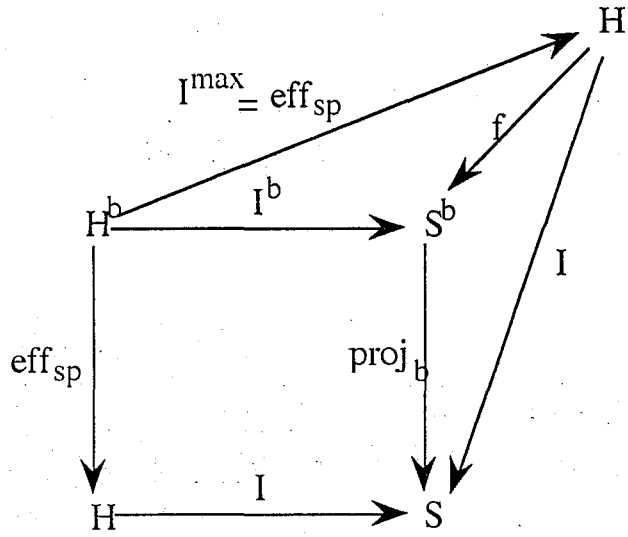

Figure 7.5: The homomorphism between the maximal PIE and a backtrack undo of the same original PIE P.

original PIE $\mathcal{P}$. The meaning of this theorem is that, even if we cannot have all information of the sets of states of the backtrack *Undo*, there is a kind of upper bound on such information and such upper bound is represented by S^{max}. We cannot have more information (in sense of reachable states) than S^{max}.

Theorem 7.1 *Given a PIE $\mathcal{P}^b$ and the PIE $\mathcal{P}^{max}$, both of them backtrack Undo of the same original PIE $\mathcal{P}$, then there exists a homomorphism f, $f : S^{max} \rightarrow S^b$, such that the diagram of Figure 7.4 commutes.*

Proof: Considering the definition of $\mathcal{P}^{max}$, we can redraw the diagram of Figure 7.4 as in Figure 7.5. The last commutes if it commutes for any path starting from the source node H^b and arriving to the target node S. This means that we have to prove that the square and both the triangles commute. Since $\mathcal{P}^b$ is a backtrack *Undo* of $\mathcal{P}$, then the square commutes. Now, we can redraw the two triangles into the square of Figure 7.6. Such a square, without the f function, commutes because, due to the fact that both $\mathcal{P}^{max}$ and $\mathcal{P}^b$ are backtrack *Undo* of the same PIE $\mathcal{P}$, we have that $\forall h \in H^b \bullet I(eff_{sp}(h)) = proj^b(I^b(h))$. When we add the f function, we don't need to prove the commutativity of both the triangles, but only of the upper-left one. In fact, if the last commutes, since eff_{sp} is surjective, then, by exploting the composition of functions and the commutativity of the square, we can easily derive the commutativity of the lower-right triangle. So, our thesis becomes to prove that $\forall h \in H^b \bullet I^b(h) = f(eff_{sp}(h))$. In the upper-left triangle, the f function represents a $0 - morphism$ between $\mathcal{P}^{max}$ and $\mathcal{P}^b$.
To prove this theorem, we need to introduce a function, $nat : H \rightarrow H^b$, which is the natural injection of H in H^b. Such a *nat* function is the right inverse of eff_{sp}, so that $eff_{sp} \circ nat = id$. We assert that $f = I^b \circ nat$ is a suitable function producing the required 0-morphism between $\mathcal{P}^{max}$ and $\mathcal{P}^b$. Considering this choice of f and the definition of I^{max}, the thesis of this theorem becomes to prove that

$\forall h \in H^b \bullet I^b(h) = I^b(nat(eff_{sp}(h)))$,

that is the diagram of Figure 7.6 commutes. Also this proof is done by induction. We consider four different cases:

(a) $h = <>$;

(b) $h = h' \frown c$;

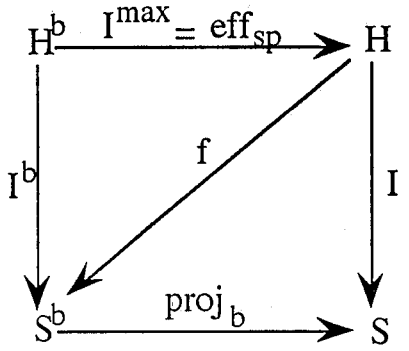

Figure 7.6: The 0-morphism f.

(c) $h = Undo^n$;

(d) $h = h_k \frown c \frown Undo^{n-k-1}$.

(a) $h =<>$.

Left

By definition of I^b we have that $I^b(<>) = s_0^b$.

Right

By definition of eff_{sp} we have that:

$$I^b(nat(eff_{sp}(<>))) = I^b(nat(<>))$$

Since nat is the natural injection of H in H^b, then we have:

$$I^b(nat(<>)) = I^b(<>)$$

which, by definition of I^b, is equal to s_0^b.

$\Box(a)$

(b) $h = h' \frown c$.

Assume that $I^b(h) = I^b(nat(eff_b(h)))$, prove that $I^b(h \frown c) = I^b(nat(eff_b(h \frown c)))$.

Left

By definition of I^b we have that $I^b(h \frown c) = doit^b(I^b(h), c)$.

Right

By definition of eff_b we have that

$$I^b(nat(eff_{sp}(h \frown c))) = I^b(nat(eff_{sp}(h) \frown c))$$

Similarly, by definition of nat, we have that

$$I^b(nat(eff_{sp}(h) \frown c)) = I^b(nat(eff_{sp}(h)) \frown c)$$

By definition of I^b we have that:

$$I^b(nat(eff_{sp}(h) \frown c)) = doit^b(I^b(nat(eff_{sp}(h)), c))$$

By applying inductive hypothesis on the I^b, we have:

$$doit^b(I^b(nat(eff_{sp}(h)), c)) = doit^b(I^b(h), c))$$

so, the left-hand side and right-hand side are equal.

$\Box(b)$

(c) $h = Undo^n$.

We need to prove that $I^b(Undo^{n+1}) = I^b(nat(eff_{sp}(Undo^{n+1})))$. Left

By Property P3 of Proposition 6.2, we have that $I^b(Undo^{n+1}) = s_0^b$.

Right

By Property P1 of Proposition 6.1, we have that $I^b(nat(eff_{sp}(Undo^{n+1}))) = I^b(nat(<>))$
which, by definition of nat is

$$= I^b(<>)$$

and the last, by definition of I^b, is equal to s_0^b. So, the left-hand side and right-hand side are equal.

$\Box(c)$

(d) $h = h_k \frown c \frown Undo^{n-k-1}$.

Assume that the inductive hypothesis holds for all h' where $length(h') \leq n$, prove that

$$I^b(h_k \frown c \frown Undo^{n-k-1} \frown Undo)$$
$$= I^b(nat(eff_{sp}(h_k \frown c \frown Undo^{n-k-1} \frown Undo))).$$

Left

By property of *Undo*, we have that

$$I^b(h_k \frown c \frown Undo^{n-k-1} \frown Undo)$$
$$= I^b(h_k \frown c \frown Undo \frown Undo^{n-k-1})$$

which, by applying equation E1, becomes equal to $I^b(h_k \frown Undo^{n-k-1})$.

Right

By applying Property P2 of Proposition 6.1, we have that

$$I^b(nat(eff_{sp}(h_k \frown c \frown Undo^{n-k-1} \frown Undo)))$$
$$= I^b(nat(eff_{sp}(h_k \frown Undo^{n-k-1})))$$

Since this sequence is shorter than n, then we can apply inductive hypothesis, obtaining:

$$I^b(nat(eff_{sp}(h_k \frown Undo^{n-k-1}))) = I^b(h_k \frown Undo^{n-k-1})$$

so, left-hand side and right hand side are equal.

□□

7.4 Existence of a homomorphism for P^{min}

As we wanted to prove that $\mathcal{P}^{max}$ is the maximal backtrack *Undo*, similarly we want to prove that $\mathcal{P}^{min}$ is the minimal one. Being the structure of $\mathcal{P}^{min}$ more complex than $\mathcal{P}^{max}$, we have to introduce functions and properties that we will use in the following proofs. We are going to start by introducing some functions useful while proving that $\mathcal{P}^{min}$ is monotone.

Proposition 7.1 *Given the function $\phi_i : H^b \to S$ such that $\forall i > 0, \phi_i(h) = I(chop_i(eff_{sp}(h)))$, then the following properties hold:*

$\phi_0(h \frown c)$	$=$	$doit(\phi_0(h), c)$	P4
$\phi_i(h \frown c)$	$=$	$\phi_{i-1}(h)$	P5
$\phi_i(h \frown Undo)$	$=$	$\phi_{i+1}(h)$	P6

Proof: The proof is done by induction.

(P4)

By definition of ϕ_i, we have that

$$\phi_0(h \frown c) = I(chop_0(eff_{sp}(h \frown c)))$$

which, by definition of $chop_0$ is equal to $I(eff_{sp}(h \frown c))$. By definition of eff_{sp}, we have that

$$I(eff_{sp}(h \frown c)) = I(eff_{sp}(h) \frown c)$$

which, by definition of I, is equal to $doit(I(eff_{sp}(h)), c)$. But, by definition of ϕ_0, we have that

$$doit(I(eff_{sp}(h)), c) = doit(\phi_0(h), c)$$

$\Box(P4)$

(P5)

By definition of ϕ_i, we have that

$$\phi_i(h \frown c) = I(chop_i(eff_{sp}(h \frown c)))$$

which, by definition of eff_{sp} is equal to $I(chop_i(eff_{sp}(h) \frown c))$.

By definition of $chop_i$, we have that

$$I(chop_i(eff_{sp}(h) \frown c)) = I(chop_{i-1}(eff_{sp}(h)))$$

which, by definition of ϕ_i, is equal to $\phi_{i-1}(h)$.

$\Box(P5)$

(P6)

By definition of ϕ_i, we have that

$$\phi_i(h \frown Undo) = I(chop_i(eff_{sp}(h \frown Undo)))$$

By definition of eff_{sp}, we have that

$$I(chop_i(eff_{sp}(h \frown Undo))) = I(chop_i(chop_1(eff_{sp}(h))))$$

which is equal to $I(chop_{i+1}(eff_{sp}(h)))$.

By definition of ϕ_i, we have that

$$I(chop_{i+1}(eff_{sp}(h)) = \phi_{i+1}(h)$$

$\Box\Box$

Interpretation function of P^{min}

In Section 7.2 $\mathcal{P}^{min}$ was defined in terms of its state update function $doit^{min}$. Although it is possible to derive the interpretation function I^{min} from that of $doit^{min}$, we will find useful to have an explicit definition of I^{min}. We therefore have to find a suitable interpretation function I^{min} and we have to prove that it corresponds to the $doit^{min}$ function. This is formulated in the following proposition:

Proposition 7.2 *Given the function* $I^{min} : H \rightarrow S^*$ *such that*

- $I^{min}(h) =< \phi_{n-1}(h), \phi_{n-2}(h), \ldots, \phi_1(h), \phi_0(h) >$, where $n \leq length(eff_{sp}(h))$;
- $\phi_{n-1}(h) \neq s_0$;
- $\phi_i(h) = s_0, \forall i \geq n$

then I^{min} *corresponds to the* $doit^{min}$.

Proof: In order to prove that I^{min} corresponds to $doit^{min}$, we have to prove that:

(a) $I^{min}(<>) = s_0^{min}$;

(b) $I^{min}(h \frown c_u) = doit^{min}(I^{min}(h), c_u)$

where c_u indicates an ordinary command $c \in C$ or an *Undo*.

(a)

Left

If $h =<>$, then $length(eff_{sp}(h)) = 0$, so $\forall i, \phi_i(<>) = s_0$. This means that $I^{min}(<>) =<>$.

Right

$s_0^{min} =<>$ by definition of s_0^{min}.

$\square(a)$

(b) For the case (b) we have to consider two subcases, depending on I^{min}. We have to consider the case in which $I^{min}(h) \neq <>$ and the other in which $I^{min}(h) = <>$. In the previous one, we have other two subcases: the first in which c_u is an ordinary command $c \in C$, the second in which c_u is an *Undo*. Moreover, when $I^{min}(h) = <>$, we have to consider the case in which we perform a command still remaining in the same state, or we perform a command and we change state, or we perform and *Undo*. This cases are skematised as follows:

(b_{a1}) $I^{min}(h) \neq <>$, $c_u = c$;

(b_{a2}) $I^{min}(h) \neq <>$, $c_u = Undo$;

(b_{b1}) $I^{min}(h) = <>$, $c_u = c$, $doit(s_0, c) = s_0$;

(b_{b2}) $I^{min}(h) = <>$ $c_u = c$, $doit(s_0, c) \neq s_0$;

(b_{b3}) $I^{min}(h) = <>$, $c_u = Undo$.

In all the three (b_b) cases, the fact that $I^{min}(h) = <>$ means that $\forall i, \phi_i(h) = s_0$.

(b_{a1})

Left

The left side is given by the enunciate of the proposition.

Right

By definition of $doit^{min}$ we have that:

$$doit^{min}(I^{min}(h), c) = I^{min}(h) \frown (doit(last(I^{min}(h)), c)$$

Since $I^{min} =< \phi_{n-1}(h), \phi_{n-2}(h), \ldots, \phi_1(h), \phi_0(h) >$, then its last element is $\phi_0(h)$. So, by applying the definition of I^{min} to the argument of the *doit*, we have:

$$I^{min}(h) \frown (doit(last(I^{min}(h)), c) = I^{min}(h) \frown (doit(\phi_0(h)), c)$$

By applying porperty P4 of Proposition 7.1, we have:

$$I^{min}(h) \frown (doit(\phi_0(h)), c) = I^{min}(h) \frown \phi_0(h \frown c)$$

which, by definition of I^{min}, becomes:

$$I^{min}(h) \frown \phi_0(h \frown c)$$
$$=< \phi_{n-1}(h), \phi_{n-2}(h), \ldots . \phi_1(h), \phi_0(h), \phi_0(h \frown c) >$$

By applying property P5 of Proposition 7.1, we have that

$$< \phi_{n-1}(h), \phi_{n-2}(h), \ldots, \phi_1(h), \phi_0(h), \phi_0(h \frown c) >= \\ < \phi_n(h \frown c), \phi_{n-1}(h \frown c), \ldots, \phi_2(h \frown c), \phi_1(h \frown c), \phi_0(h \frown c) >$$

which, by definition of I^{min}, is equal to $I^{min}(h \frown c)$.
So the left-hand side and right-hand side are equal.

$\square(b_{a1})$

(b_{a2})

Left

The left side is given by the enunciate of the proposition.

Right

By definition of $doit^{min}$, we have that:

$$doit^{min}(I^{min}(h), Undo) = chop_1(I^{min}(h))$$

By definition of I^{min}, we have that:
$chop_1(I^{min}(h)) = chop_1(< \phi_{n-1}(h), \phi_{n-2}(h), \ldots, \phi_1(h), \phi_0(h) >)$
$=< \phi_{n-1}(h), \phi_{n-2}(h), \ldots, \phi_1(h) >$
which, by applying property P6 of Proposition 7.1, is equal to

$$< \phi_{n-2}(h \frown Undo), \ldots, \phi_1(h \frown Undo), \phi_0(h \frown Undo) >$$

Finally, by applying again the definition of I^{min}, we have that:

$$< \phi_{n-2}(h \frown Undo), \ldots, \phi_1(h \frown Undo), \phi_0(h \frown Undo) > \\ = I^{min}(h \frown Undo)$$

So the left-hand side and right-hand side are equal.

$\square(b_{a2})$

(b_{b1})

Left

By definition of I^{min}, we have that

$$I^{min}(h \frown c) =< \phi_{n-1}(h \frown c), \ldots, \phi_0(h \frown c) >$$

By applying property P5 of Proposition 7.1,
we have that $\forall i > 0, \phi_i(h \frown c) = \phi_{i-1}(h)$

In this way, we can rewrite *imin* as follows:

$$I^{min}(h \frown c) =< \phi_{n-2}(h), \ldots, \phi_0(h), \phi_0(h \frown c) >$$

Since $I^{min}(h) =<>$, that is $\forall i, \phi_i(h) = s_0$, that is
$\forall i > 0, \phi_i(h \frown c) = s_0$, then *imin* is as follows:

$$I^{min}(h \frown c) =< \phi_0(h \frown c) >$$

By property P4 of Proposition 7.1 we have that
$\phi_0(h \frown c) = doit(\phi_0(h), c)$.

Since $\forall i, \phi_i(h) = s_0$, then we have that

$$doit(\phi_0(h), c) = doit(s_0, c)$$

We are in the case in which $doit(s_0, c) = s_0$, so that, clumping the last equalities involving the $doit^{min}$, we have

$$\phi_0(h \frown c) = doit(\phi_0(h), c) = doit(s_0, c) = s_0$$

This means that $\phi_i(h \frown c) = s_0$ $\forall i$, including also the case of $i = 0$. So, by definition of I^{min}, we have that:

$$I^{min}(h \frown c) =<>$$

Right

In this case, $I^{min}(h) =<>$, $\forall i, \phi_i(h) = s_0$, $c_u = c$, and $doit(s_0, c) = s_0$. So, by definition of $doit^{min}$, we have that:

$$doit^{min}(I^{min}(h), c) = doit^{min}(<>, c) =<>$$

So the left-hand side and right-hand side are equal.

$\square(b_{b1})$

(b_{b2})

Left

The proof of the left-hand side of the case (b_{b2}) is the same as the left-hand side of the case (b_{b1}) but the last step. Anyway, in order to avoid to lose one's bearing, we report the proof step by step.

By definition of I^{min}, we have that

$$I^{min}(h \frown c) =< \phi_{n-1}(h \frown c), \ldots, \phi_0(h \frown c) >$$

By applying property P5 of Proposition 7.1,
we have that $\forall i > 0, \phi_i(h \frown c) = \phi_{i-1}(h)$
In this way, we can rewrite *imin* as follows:

$$I^{min}(h \frown c) =< \phi_{n-2}(h), \ldots, \phi_0(h), \phi_0(h \frown c) >$$

Since $I^{min}(h) =<>$, that is $\forall i, \phi_i(h) = s_0$, that is
$\forall i > 0, \phi_i(h \frown c) = s_0$, then *imin* is as follows:

$$I^{min}(h \frown c) =< \varphi_0(h \frown c) >$$

By property P4 of Proposition 7.1 we have that
$\phi_0(h \frown c) = doit(\phi_0(h), c)$.
Since $\forall i, \phi_i(h) = s_0$, then we have that

$$doit(\phi_0(h), c) = doit(s_0, c)$$

that is

$$\phi_0(h \frown c) = doit(\phi_0(h), c) = doit(s_0, c)$$

Since we are in the case in which $doit(s_0, c) \neq s_0$, then we have that :

$$I^{min}(h \frown c) =< doit(s_0, c) >$$

Right

In this case, $I^{min}(h) =<>$, $\forall i, \phi_i(h) = s_0$, $c_u = c$, and $doit(s_0, c) \neq s_0$.
By definition of $doit^{min}$, we have that:

$$doit^{min}(I^{min}(h), c) = doit^{min}(<>, c) =< doit(s_0, c) >.$$

So the left-hand side and right-hand side are equal.

$\square(b_{b2})$

(b_{b3})

This is the case in which $I^{min}(h) =<>$, which implies that $\forall i, \phi_i(h) = s_0$ and $c_u = Undo$.

Left

By definition of I^{min}, we have that

$$I^{min}(h \frown Undo) =< \phi_{n-1}(h \frown Undo), \ldots, \phi_0(h \frown Undo) >$$

By applying property P6, we have that

$$I^{min}(h \frown c) =< \phi_n(h), \ldots, \phi_1(h) >$$

but, since we are in the case in which $\forall i, \phi_i(h) = s_0$, then $I^{min}(h \frown c) =<>$

Right

Since $I^{min}(h) =<>$, then we have that

$$doit^{min}(I^{min}(h), Undo) = doit^{min}(<>, Undo)$$

which is equal to $<>$ by definition of $doit^{min}$.
So the left-hand side and right-hand side are equal.

□□

7.4.1 A new kind of interpretation function

The I^{min} function introduced in the previous Section associates a sequence of states to any history given as input. But such a sequence is starting from a state which is different from the initial one. In fact, S^{min} is "smaller", with respect to the sets of states of other backtrack *Undo* of the same original PIE $\mathcal{P}$, because it does not consider the initial state, and any its repetition, at the beginning of the sequence. Since the difference between $\mathcal{P}^{min}$ and any other backtrack *Undo* of the same original PIE $\mathcal{P}$ is exactly in the fact that in $\mathcal{P}^{min}$ the sequence of states is chopped when at the beginning we have a repetition of the initial state, in order to establish a relationship betwen S^{min} and the set of states of another backtrack *Undo*, we need to introduce another interpretation function. Such function should consider also the case of sequences of states infinite to the left, that is with any number of initial state at the beginning. At this aim we introduce the following function:

$$strip : S_0^{-\infty} \rightarrow S^*$$

$strip(hs)$	$=<>$	if $h_s = s_0^*$
$strip(h_s \frown s)$	$= strip(h_s) \frown s$	if $s \neq s_0$
$strip(h_s \frown s_0)$	$=<>$	if $strip(h_s) =<>$
$strip(h_s \frown s_0)$	$= strip(h_s) \frown s_0$	otherwise

where $S_0^{-\infty}$ is the set of the sequences of states $s \in S$, sequences infinite to the left, in which the infinity is given by a sequence of any number of initial state.

Proposition 7.3 *Given the strip function as above defined, the following property holds:*

$$strip(s_0^* \frown h_s) = \begin{cases} <> & \text{if } h_s = <> \\ h_s & \text{if } first(h_s) \neq s_0 \end{cases}$$

Proof: To prove this property, we have to consider the case in which the sequence of state h_s is empty, $h_s = <>$, or not. In the second case, at least the first element of h_s is different from the initial state. These two cases are skematised as follows:

(1) $h_s = <>$;

(2) $h_s = s \frown h_s\prime, s \neq s_0$.

(1)

By definition of *strip*, we have that:

$$strip(s_0^* \frown <>) = strip(s_0^*) = <>$$

(2)

The proof of (2) is done by induction on h_s'.

(2i) base case: $h_s = <>$

By definition of *strip* we have that:

$$strip(s_0^* \frown s \frown h_s') = strip(s_0^* \frown s) = strip(s_0^*) \frown s = <> \frown s = < s >.$$

(2ii) assume valid: $strip(s_0^* \frown s \frown h_s') = s \frown h_s'$, prove that $strip(s_0^* \frown s \frown h_s' \frown s') = s \frown h_s' \frown s'$.

By definition of *strip*, we have that

$strip(s_0^* \frown s \frown h_s' \frown s') = strip(s_0^* \frown s \frown h_s') \frown s'$.

Now, we can apply inductive hypothesis on the argument of the *strip*, obtaining $s \frown h_s' \frown s'$. Such an equality holds independently if s' is the initial state or another one.

□□

Now we can introduce an interpretation function which associates to any input a sequence of states infinite to the left.

Lemma 7.3 *Given the function*

$$I'(h) : H^b \rightarrow S_0^{-\infty}$$
$$I'(h) = s_0^* \frown I^{min}(h)$$

then $\forall h \in H^b, I^{min}(h) = strip(I'(h))$.
Proof: The proof follows directly from the properties of the *strip* function and the definition of I^{min}.

□□

Before to enunciate the theorem of existence of a homomorphism for $\mathcal{P}^{min}$, we need to introduce other functions and properties. Indicating with $S^{-\infty}$ the set of sequences infinite to the left, we can introduce the following function:

$$g' : S^b \rightarrow S^{-\infty}$$
$$g'(s) = < \ldots, g_i(s), \ldots, g_0(s) >$$
where every g_i is so defined:
$$g_i : S^b \rightarrow S$$
$$g_i(s) = proj^b(k_i(s))$$
where $k_i = S^b \rightarrow S^b$
$$k_0(s) = s$$
$$k_i(s) = doit^b(k_{i-1}(s), Undo)$$

The formal functions g_i and k_i emulate the effect of someone experimenting with a system by using the *Undo* function. From a given starting state s, $k_i(s)$ gives the state of the system after performing i times *Undo* and $g_i(s)$ gives the projected state, that is the state of the underlying system without *Undo*.

Proposition 7.4 *Given the function k_i above introduced, the two following properties hold:*

(1) $k_{i+1}(s) = k_i(doit^b(s, Undo))$; P7

(2) $\forall i, s_0^b$ is fixed point for k_i. P8

Proof: For both the properties, P7 and P8, the proof is done by induction.

P7

• base case: $i = 0$.

Left

By definition of k_i, we have that:
$k_1(s) = doit^b(k_0(s), Undo) = doit^b(s, Undo)$.

Right

By definition of k_0, we have that
$k_0(doit^b(s, Undo)) = doit^b(s, Undo)$
so the left-hand side and right-hand side are equal.

•• Assume that $k_{i+1}(s) = k_i(doit^b(s, Undo))$, prove that
$k_{i+2}(s) = k_{i+1}(doit^b(s, Undo))$

Left

By definition of k_i, we have that
$k_{i+2}(doit^b(s, Undo)) = doit^b(k_{i+1}(s), Undo)$.
By applying induction to the argument of the $doit^b$, we have that:
$= doit^b(k_i(doit^b(s, Undo)), Undo)$.

Right

By definition of k_i, we have that:
$k_{i+1}(doit^b(s, Undo)) = doit^b(k_i(doit^b(s, Undo)), Undo)$
so the left-hand side and right-hand side are equal.

□*P7*

P8

To prove that $\forall i.\ s_0^b$ is fixed point for k_i, means to prove that $\forall i, k_i(s_0^b) = s_0^b$.

• base case, $i = 0$.

By definition of k_0, we have that $k_0(s_0^b) = s_0^b$.

•• Assume that $k_i(s_0^b) = s_0^b$, prove that $k_{i+1}(s_0^b) = s_0^b$.

By definition of k_i, we have that
$k_{i+1}(s_0^b) = doit^b(k_i(s_0^b), Undo)$.

By applying inductive hypothesis to the argument of the $doit^b$, the right element of the above written equality is equal to $doit^b(s_0^b, Undo)$. Since $\mathcal{P}^b$ is backtrack *Undo* of the PIE $\mathcal{P}$, then equation E1 holds, so the last is equal to s_0^b.

□*P8*

The k_i function was introduced in order to define g', but we have not yet introduced properties regarding the function g' itself. g' is a function which associates with a state $s \in S^b$ a sequence of states infinite to the left. These are the states that would be obtained by applying *Undo* repeatedly to the original state s. Remember that we are only interested in reachable states, that is those in the range of the interpretation function I^b. This means that we only want to talk about those states $s \in S^b$ which are given by the interpretation of a suitable history $h \in H^b$.

What we want to prove now, is that the infinite part of each g' is given by a sequence of any number of initial state s_0^b. This is expressed in the following proposition:

Proposition 7.5 *Given the g' function, so defined*

$$
\begin{array}{l}
g' : S^b \rightarrow S^{-\infty} \\
\quad g'(s) = < \ldots, g_i(s), \ldots, g_0(s) > \\
\text{where each } g_i \text{ is so defined:} \\
\quad g_i : S^b \rightarrow S \\
\qquad g_i(s) = proj^b(k_i(s)) \\
\qquad \text{where } k_i(s) = S^b \rightarrow S^b \\
\qquad\quad k_0(s) = s \\
\qquad\quad k_i(s) = doit^b(k_{i-1}(s), Undo)
\end{array}
$$

then $\forall s \in I^b(H^b)\ \exists n \in N$ **st.** $\forall i > n\ g_i(I^b(h)) = s_0$.

Proof: Before to start the proof, we have to do a couple of consideration. First, considering that the I^b function is surjectiveon $I^b(H^b)$, we could rephrase the enunciate of the proposition as follows:

$\forall h \in H^b\ \exists n \in N$ **st.** $\forall i > n,\ g_i(I^b(h)) = s_0$.

Second, if we observe such an enunciate, we can see two symbols $\forall$, one for i and the other for h. This means that we have to do two inductive proofs, one on n an the other on h. We can choose $n = length(h)$.Note, we do not claim that $length(h)$ is the minimal n, just that it satisfies the conditions for Proposition 7.5. We are going to start the proof working on i.

Since $g_i(I^b(h)) = proj^b(k_i(I^b(h)))$, then, for the conservativeness of state, if $k_i(I^b(h)) = s_0^b$, we have $g_i(I^b(h)) = proj^b(s_0^b) = s_0$ and the proof is done. So we have to work on $k_i(I^b(h))$.

Induction on i

Given an $h \in H^b$, let $n = length(h)$. We will assume that $k_n(I^b(h)) = s_0^b$ (this will be proved below by induction on n). Given this assumption, we shall prove that $\forall\ i > n\ , k_i(I^b(h)) = s_0^b$. In fact, we will prove that given the choice $n = length(h)$ we have the stronger result that:

$\forall i > n,\ k_i(I^b(h)) = s_0^b$.

It is sufficient to prove this as $g_i(I^b(h)) = proj^b(k_i(I^b(h)))$, and so, because of the conservativeness of state, if $k_i(I^b(h)) = s_0^b$ we also have $g_i(I^b(h)) = proj^b(s_0^b) = s_0$ and the proof is done.

• base case, $i = n$. It is given by definition.

•• General case.
Assume that $k_i(I^b(h)) = s_0^b$. Prove that $k_{i+1}(I^b(h)) = s_0^b$.

By definition of k_i, we have that

$$k_{i+1}(I^b(h)) = doit^b(k_i(I^b(h)), Undo)$$

By applying the inductive hypothesis to the argument of the $doit^b$, we have :

$$doit^b(k_i(I^b(h)), Undo) = doit^b(s_0^b, Undo)$$

which is equal to s_0^b since equation E4 holds.

Induction on h

Let $n = length(h)$ prove that $\forall h\ \in H^b\ k_n(I^b(h)) = s_0^b$.

• base case $h =<>$

Since $n = length(h)$, if $h =<>$, then in the base case we have that $n = 0$. So we have to prove that $k_0(I^b(<>)) = s_0^b$.

By definition of k_i, we have that:

$$k_0(I^b(<>)) = k_0(s_0^b) = s_0^b$$

•• Assume that $k_n(I^b(h)) = s_0^b$, prove that $k_{n+1}(I^b(h \frown c)) = s_0^b$.

Note that, by increasing the length of h, from h to $h \frown c$, also n increases from n to $n + 1$. Moreover, since for Theorem 7.1 (existence of a homomorphism for $\mathcal{P}^{max}$) we have that $I^b(h) = I^b(nat(eff_{sp}(h)))$, and $nat(eff_{sp}(h)) \in II \subset II^b$, it is sufficient to do the proof $\forall h \in H$, so we can don't consider the case in which $c = Undo$.

By applying property P7 of Proposition 7.4, we have that

$$k_{n+1}(I^b(h \frown c)) = k_n(doit^b(I^b(h \frown c), Undo))$$

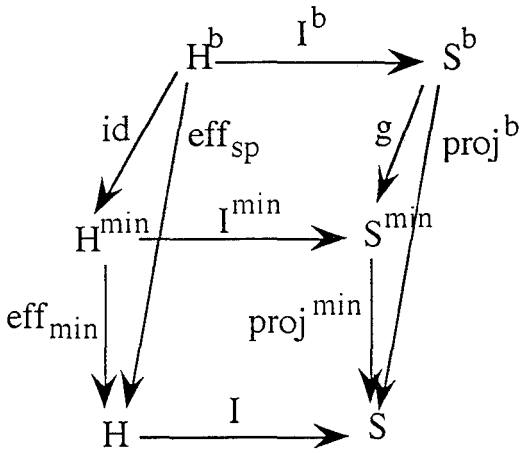

Figure 7.7: Encapsulation of the minimal PIE and of a backtrack undo of the same original PIE P.

Since $\mathcal{P}$ is monotone, we can express the $doit^b$ in terms of the I^b and, since equation E1 holds, we have:

$$k_n(doit^b(I^b(h \frown c), Undo)) = k_n(I^b(h))$$

By applying inductive hypothesis, we have: $k_n(I^b(h)) = s_0^b$.

□□

7.5 Existence of a homomorphism for P^{min}

Besides the relationship between the set of states of $\mathcal{P}^{max}$ and any backtrack *Undo* $\mathcal{P}^b$, we could also find a similar relationship between the set of states of any backtrack *Undo* $\mathcal{P}^b$ and $\mathcal{P}^{min}$, both of them backtrack *Undo* of the same PIE $\mathcal{P}$. The meaning of such relationship is that S^{min} represents the "smallest" set of states which still allows the user to perform backtrack *Undo*. In some sense, we can see S^{min} as a lower bound of the set of states of any

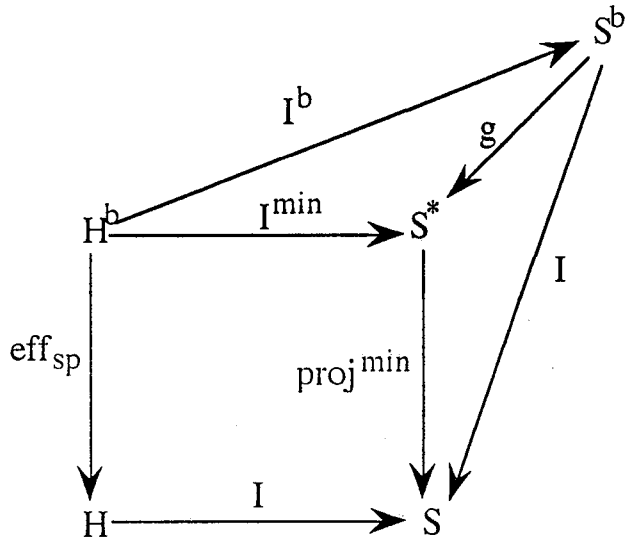

Figure 7.8: The homomorphism between the minimal PIE and a backtrack undo of the same original PIE P.

backtrack *Undo*. We cannot have less information (in sense of reachable states) than S^{min}. The above mentioned relationship is formalized in the following theorem:

Theorem 7.2 *Given a PIE $\mathcal{P}^b$ and the PIE $\mathcal{P}^{min}$, both of them backtrack Undo of the same original PIE $\mathcal{P}$, then there exists a homomorphism g, $g : S^b \rightarrow S^{min}$, such that the diagram of Figure 7.7 commutes.*

Proof: Considering the definition of $\mathcal{P}^{min}$, we can redraw the diagram of Figure 7.7 as in Figure 7.8. The last commutes if it commutes for any path starting from the source node H^b and arriving to the target node S. This means that we have to prove that the square and both the triangles commute. Since $\mathcal{P}^{min}$ is a backtrack *Undo* of $\mathcal{P}$, then the square commutes. Now, we can redraw the two triangles into the square of Figure 7.9. Such a square, without the g function, commutes because, due to the fact that both $\mathcal{P}^{min}$ and $\mathcal{P}^b$ are backtrack *Undo* of the same PIE $\mathcal{P}$, we have that $\forall h \in H^b \bullet proj^{min}(I^{min}(h)) = I(eff_{sp}(h)) = proj^b(I^b(h))$. Following the same reasoning as in the case of the existence of a homomorphis between S^{max} and S^b, when we add the g function, it is sufficient to prove the commutativity only of the upper-left triangle, that is we have to prove that

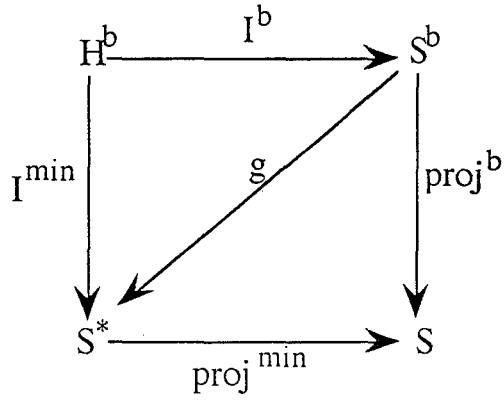

Figure 7.9: The 0-morphism g.

$\forall h \in H^b$, $I^{min}(h) = g(I^b(h))$. Similarly for $\mathcal{P}^{max}$, the g function represents a $0-morphism$, this time between $\mathcal{P}^b$ and $\mathcal{P}^{min}$. We choose our g function as follows: $g : S^b \rightarrow S^*$ such that

$$g(s) = \begin{cases} strip(g'(s)) & \text{if } s \in I^b(H^b) \\ <> & \text{otherwise} \end{cases}$$

Note that the second part of the definition covers unreachable states. It is purely to make the function g total and the value $<>$ is arbitrary.

In Theorem 7.1 we proved that for any $\mathcal{P}^b$ backtrack *Undo* of of a PIE $\mathcal{P}$, it is possible to find a homomorphism between S^{max} and S^b. We proved this by introducing the *nat* function, $nat : H \rightarrow H^b$, which is the right inverse of the ϵff_{sp}. Since $\mathcal{P}^b$ may be *any* backtrack *Undo* of the same PIE $\mathcal{P}$, then there exists a homomorphism also from S^{max} to S^{min}. By applying Theorem 7.1 to $\mathcal{P}^b$ and $\mathcal{P}^{min}$ respectively, we have the two homomorphisms $I^b \circ nat$ (for $\mathcal{P}^b$), and $I^{min} \circ nat$ (for $\mathcal{P}^{min}$) such that the diagram of Figure 7.10 commutes. Considering that *nat* is the natural injection of H in H^b, and that it is the identity on $H \subset H^b$, we can restrict our application domain to any $h \in H$ instead of H^b. So, our thesis becomes to prove that

$\forall h' \in H$, $g(I^b(nat(h'))) = I^{min}(nat(h'))$
that is $\forall h \in H \subset H^b$, $g(I^b(h)) = I^{min}(h)$.
Since $\forall s \in I^b(H^b)$, $g(s) = strip(g'(s))$ and
$\forall h \in H^b$, $I^{min}(h) = strip(I'(h))$, in order to prove that $\forall h \in H \subset H^b$, $g(I^b(h)) = I^{min}(h)$ it is sufficient to prove that
$\forall h \in H$, $g'(I^b(h)) = I'(h)$.

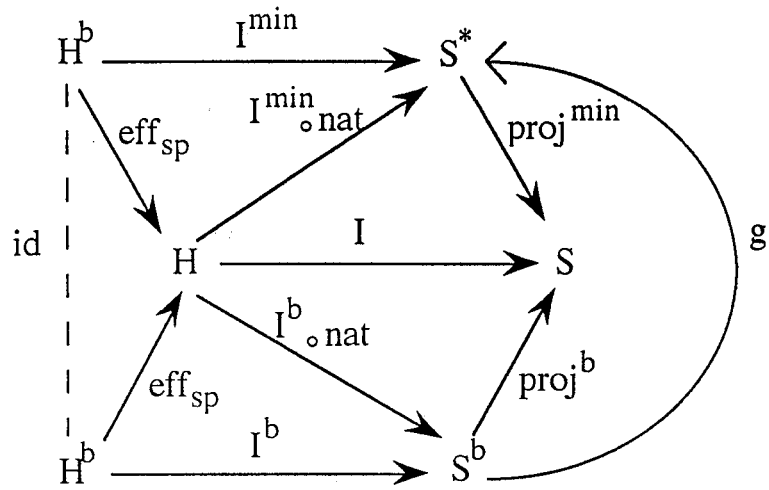

Figure 7.10: Another diagram of the 0-morphism g.

Moreover, since

$$g'(I^b(h)) =< \ldots, g_i(I^b(h)), \ldots, g_1(I^b(h)), g_0(I^b(h)) >;$$
$$I'(h) =< \ldots, \phi_i(h), \ldots, \phi_1(h), \phi_0(h) >;$$
$$\phi_i(h) = I(chop_i(eff_{sp}(h))),$$

we need to prove that $\forall i, \forall h \in H,\ g_i(I^b(h)) = \phi_i(h)$.

Let us to do some other transformation before to start the inductive proof. Since eff_{sp} is the identity on H, then $I(chop_i(eff_{sp}(h)) = I(chop_i(h))$.

Furthermore, by definition of g_i, we have that $g_i(I^b(h)) = proj^b(k_i(I^b(h)))$. So, we have to prove that:

$$proj^b(k_i(I^b(h))) = I(chop_i(h)).$$

The proof is done by applying multiple induction, mathematical induction on i and structural induction on h. We have two base cases (one in which $h =<>$ and i is whatever, the other in which $i = 0$ and h is whatever) and a general one (in which, fixed any i and fixed any h for which the hypothesis hold, we have to prove that it holds also for $i + 1$ and $h \frown c$). If we represent in a Cartesian plane the application domain of the theorem's thesis, we can label one axis with i and the other with h, then the two base

cases correspond to the proof of the thesis along the axes, while the general case corresponds to prove it in the first quadrant of such a plane, moving along any diagonal.

Now we can start the proof.

• First base case, $h = <>$, $\forall i$.
We have to prove that $proj^b(k_i(I^b(<>))) = I(chop_i(<>))$.

Left By definition of $I^b(h)$, we have that

$$proj^b(k_i(I^b(<>))) = proj^b(k_i(s_0^b))$$

By applying property P8, since s_0^b is fixed point for k_i, we have that:

$$proj^b(k_i(s_0^b)) = proj^b(s_0^b)$$

which is equal to s_0 by applying conservativeness of the state.

Right By definition of $chop_i$, we have that

$$I(chop_i(<>)) = I(<>)$$

which is equal to s_0 by definition of I. So left-hand side and right-hand side are equal.

□

•• Second base case, $i = 0$, $\forall h$.
We have to prove that $proj^b(k_0(I^b(h))) = I(chop_0(h))$.
Left By definition of k_0 we have that

$$proj^b(k_0(I^b(h))) = proj^b((I^b(h))$$

By applying condition C1 of the encapsulation, we have that

$$proj^b((I^b(h)) = I(\epsilon ff_{sp}(h))$$

which is equal to $I(h)$ because ϵff_{sp} is the identity on H.

Right
By definition of $chop_0$ we have that $I(chop_0(h)) = I(h)$.
So left-hand side and right-hand side are equal.

□

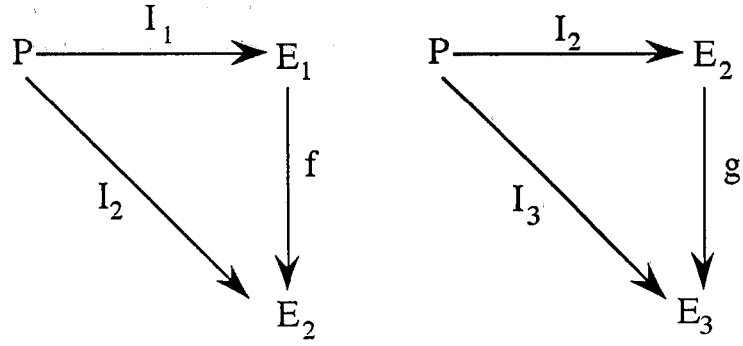

Figure 7.11: Two 0-morphism f and g.

• • • General case. Assume that $proj^b(k_i(I^b(h))) = I(chop_i(h))$, prove that $proj^b(k_{i+1}(I^b(h \frown c))) = I(chop_{i+1}(h \frown c))$.

Left Since $\mathcal{P}^b$ is monotone, we can express the the interpretation function in terms of the $doit^b$, that is:

$$proj^b(k_{i+1}(I^b(h \frown c))) = proj^b(k_{i+1}(doit^b(I(h), c)))$$

By applying property P7 of Proposition 7.4, we have that:

$$proj^b(k_{i+1}(doit^b(I(h), c))) = proj^b(k_i(doit^b(doit^b(I(h), c), Undo)))$$

The last, by equation E1, is equal to $proj^b(k_i(I^b(h)))$. Now, by applying inductive hypothesis, we have:

$$proj^b(k_i(I^b(h))) = I(chop_i(h))$$

Right
By definition of $chop_i$ we have that:

$$I(chop_{i+1}(h \frown c)) = I(chop_i(h))$$

So the left-hand side and right-hand side are equal.

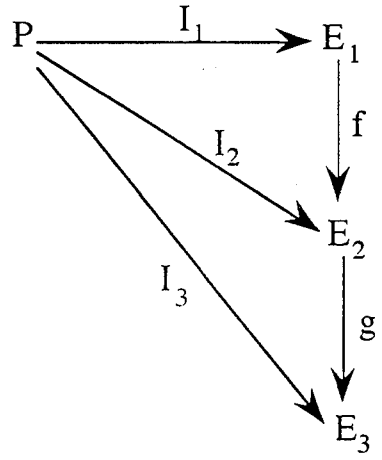

Figure 7.12: Composition of two 0-morphism.

7.6 Categorical representation of backtrack Undo

The 0-morphisms between $\mathcal{P}^{max}$ and any $\mathcal{P}^b$ and between any $\mathcal{P}^b$ and $\mathcal{P}^{min}$ establish a kind of partial order starting from the richer set S^{max} and arriving to the "smaller" set S^{min}. The graphical representation of such homomorphisms with arrows suggests us that the class of all the backtrack *Undo* of the same PIE $\mathcal{P}$ may be a category under 0-morphism, while the partial order suggest us that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ may be respectively the initial and terminal objects in such a category. In order to prove these "suggestions", we need to use some theorems and lemmas that we are going to introduce. We start with the following Lemma:

Lemma 7.4 (Composition of 0-morphisms) *The composition of two 0-morphisms between PIE is still a 0-morphism.*

Proof: Consider two PIEs $P1 =< P, I_1, E_1 >$, $P2 =< P, I_2, E_2 >$ and the two 0-morphisms $f : E_1 \rightarrow E_2$ and $g : E_2 \rightarrow E_3$ such that the two diagrams of Figure 7.11 commute. In order to prove that the diagram of Figure 7.12, which represents the composition of the two 0-morphism commutes, we have to prove that $\forall x \in P, I_3(x) = g(f(I_1(x)))$.
The right-hand side of this equation, by using the commutativity of the left-

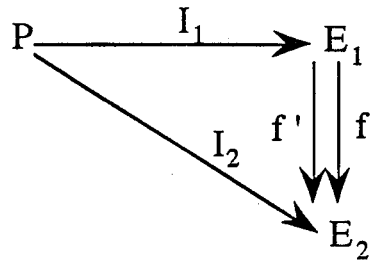

Figure 7.13: Uniqueness of 0-morphism between PIE on the reachable effects.

hand side diagram of Figure 7.11, is equal to $g(I_2(x))$, which, by using the commutativity of the other diagram of Figure 7.11, is equal to $I_3(x)$.

□□

Lemma 7.5 (Uniqueness of 0-morphisms) *Two 0-morphisms between PIE are unique on the reachable effects.*

Proof: Consider the two PIEs $P1 =< P, I_1, E_1 >$,$P2 =< P, I_2, E_2 >$ and the two 0-morphisms $f, f' : E_1 \rightarrow E_2$. We want to prove that f and f' are coincident on the reachable effects. Since f, f' are 0-morphisms, then the diagram of Figure 7.13 commutes, that is $f \circ I_1 = I_2 = f' \circ I_1$. This means that $\forall p \in P, f(I_1(p)) = f'(I_1(p))$, that is $\forall e \in I_1(p), f(e) = f'(e)$, which implies that $f \equiv f'$ on the range of I_1.

□□

Considering the properties of 0-morphisms introduced with the two previous Lemmas, Theorem 7.1 and Theorem 7.2 on the existence of 0-morphisms between backtrack *Undo* PIEs, we can enunciate two theorems (one for $\mathcal{P}^{max}$ and the other for $\mathcal{P}^{min}$) in which the characteristics of maximality/minimality are expressed in terms of existence and unicity of 0-morphisms between PIEs:

Theorem 7.3 (Maximality of $\mathcal{P}^{max}$) *$\mathcal{P}^{max}$ is maximal in the class of the backtrack* Undo *of the same PIE $\mathcal{P}$, in the sense that, given any backtrack* Undo *$\mathcal{P}^b$ for $\mathcal{P}$, there exists a unique 0-morphism from $\mathcal{P}^{max}$ to $\mathcal{P}^b$.*

Proof: The existence of such 0-morphism is given by Theorem 7.1; the uniqueness is ensured by Lemma 7.4.

□□

Theorem 7.4 (Minimality of $\mathcal{P}^{min}$) *$\mathcal{P}^{min}$ is minimal in the class of the backtrack* Undo *of the same PIE $\mathcal{P}$, in the sense that, given any backtrack* Undo *$\mathcal{P}^b$ for $\mathcal{P}$, there exists a unique 0-morphism from $\mathcal{P}^b$ to $\mathcal{P}^{min}$ up to equivalence on the reachable effects.*

Proof: The existence of such 0-morphism is given by Theorem 7.2; the uniqueness is ensured by Lemma 7.4.

□□

The successive step is to represent the characteristics of maximality/minimality in terms of category. In order to do this, we have to introduce the definitions of graph and category [77, 7].

Definition 7.1 (Graph) *A* Graph *$\mathcal{G}$ is a quadruple $\mathcal{G} = < N, E, s, t >$ where N is a class of nodes, E is a class of edges, and s, t are two mappings $s, t, : E \rightarrow N$, called source and target respectively, such that*
$\forall f \in E,\ f : a \rightarrow b,\ a, b \in N,\ s(f) = a$ and $t(f) = b$.

Definition 7.2 *A* Category *$\mathcal{C}$ is a triple $\mathcal{C} = < \mathcal{G}_c, \_ \circ \_, id\_ >$ where $\mathcal{G}_c$ is a graph $\mathcal{G}_c =< O, A, s, t >$ in which the nodes are called objects and the edges are called arrows; and $\_ \circ \_$ and $id\_$ are the two following mappings*
1) composition $\_ \circ \_ : A \times A \rightarrow A$
2) $id\_ : O \rightarrow A$
where the composition is defined for any pair of arrows $f : a \rightarrow b$ and $g : b \rightarrow c$, giving a result $g \circ f : a \rightarrow c$. $id\_$ is defined $\forall o \in O$ giving an arrow $id_o : o \rightarrow o$, and the following properties hold:
a) $(h \circ g) \circ f = h \circ (g \circ f)$ where $h : c \rightarrow d$;
b) $f \circ id_a = f = id_b \circ f$.

In the following, we are going to introduce the definition of initial and terminal object in a category. We want to prove that the class of all the backtrack *Undo* of the same PIE $\mathcal{P}$ is a category and that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the initial and terminal object respectively. The meaning of this theorem is that, even if we cannot know everything on the states, due to the fact the

interaction is non-deterministic (non-determinism also due to *Undo*), we can try to "approximate" the set of states with a "bigger" and "smaller" set. We cannot say anything on the relationships between two general backtrack *Undo* of the same original PIE $\mathcal{P}$, but, for any of them, we can find a kind of upper and lower bound.

Definition 7.3 (Initial object) *An initial object in a category is an object a, such that for any object b, there is a unique arrow $f : a \rightarrow b$.*

Definition 7.4 (Terminal object) *A terminal object in a category is an object b, such that for any object a, there is a unique arrow $g : a \rightarrow b$.*

Theorem 7.5 *The class of all the backtrack* Undo *of the same PIE $\mathcal{P}$ forms a category under 0-morphism, the initial and terminal objects of which are respectively $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$.*

Proof: In order to prove that the class of all the backtrack *Undo* of the same PIE $\mathcal{P}$ is a category under 0-morphism, we have to:

(i) define a graph $\mathcal{G}_c =< O, A, s, t >$;

(ii) define the mapping composition;

(iii) define the mapping $id\_$;

(iv) prove the associativity for composition;

(v) prove the identity for composition.

We start by defining the graph $\mathcal{G}_c =< O, A, s, t >$:

(i)

- O = the class of $\mathcal{P}^b$ backtrack *Undo* of the same original PIE $\mathcal{P}$;
- A = equivalence classes of 0-morphisms, that is an arrow is an equivalence class $[f]$ under the following equivalence:
given $f : PIE_1 \rightarrow PIE_2$, $f' : PIE_1 \rightarrow PIE_2$, then
$f \equiv f' \; if \; f(x) = f'(x), \; \forall x \in \; range(I_1)$;
- source and target mappings s, t are so defined:
$\forall f : PIE_1 \rightarrow PIE_2, \; then \; s([f]) = PIE_1, t([f]) = PIE_2$.

$\square(i)$

(ii) Given the 0-morphisms $f : PIE_1 \to PIE_2$, $g : PIE_2 \to PIE_3$ and the composition $g \circ f : PIE_1 \to PIE_3$, we define the composition of arrows as $[g] \circ [f] = [g \circ f]$.
Now we have to prove that such a definition is well formed, that is if $g' \equiv g$ and $f' \equiv f$, then $g' \circ f' \equiv g \circ f$.
Since $g' \equiv g$ and $f' \equiv f$, then g' and f' are both 0-morphisms. For Lemma 7.4, the composition of 0-morphism is still a 0-morphism, so $g' \circ f'$ is a 0-morphism from PIE_1 to PIE_3. Moreover, by applying Lemma 7.5 (uniqueness of 0-morphism), we have that $g' \circ f' \equiv g \circ f$.

$\square(ii)$

(iii) Now we can define the identity:
Given any $PIE \in O$, we define $id_{PIE} = [id_E]$.
Clearly, id_E forms a 0-morphism, so id_{PIE} is an arrow with the given PIE as source and target.

$\square(iii)$

(iv) In order to prove the associativity for composition, we have to prove that
$[h] \circ ([g] \circ [f]) = ([h] \circ [g]) \circ [f]$.

Left
$[h] \circ ([g] \circ [f]) = [h] \circ [g \circ f] = [h \circ (g \circ f)] = [(h \circ g) \circ f]$

Right
$([h] \circ [g]) \circ [f] = ([h \circ g]) \circ [f]$
so the left-hand side and right hand side are equal.

$\square(iv)$

(v) Given $[f] : PIE_1 \to PIE_2$, in order to prove the identity for composition, we have to prove that,
$[f] \circ id_{PIE_1} = [f] = id_{PIE_2} \circ [f]$.

Left
$[f] \circ id_{PIE_1} = [f] \circ [id_{PIE_1}] = [f \circ id_{PIE_1}] = [f]$

Right
$id_{PIE_2} \circ [f] = [id_{PIE_2}] \circ [f] = [id_{PIE_2} \circ f] = [f]$ so the left-hand side and right hand side are equal.

$\square(v)$

To finish the proof of the theorem, we have to prove that $\mathcal{P}^{max}$ is the initial object and $\mathcal{P}^{min}$ is the terminal object. Because of Theorem 7.1, $\forall \mathcal{P}^b \in O$, $\exists$ 0-morphism h**st.** $h : \mathcal{P}^{max} \rightarrow \mathcal{P}^b$. Such 0-morphism is unique for Lemma 7.5. Therefore there is a unique arrow $[h] : \mathcal{P}^{max} \rightarrow \mathcal{P}^b$, that is $\mathcal{P}^{max}$ is initial.

Because of Theorem 7.2, $\forall \mathcal{P}^b \in O \; \exists$ 0-morphism h'**st.** $h : \mathcal{P}^b \rightarrow \mathcal{P}^{min}$ up to equivalence on reachable states. Such 0-morphism is unique for Lemma 7.5. Therefore there is a unique arrow $[h'] : \mathcal{P}^b \rightarrow \mathcal{P}^{min}$, that is $\mathcal{P}^{min}$ is terminal.

$\square\square$

Corollary 7.1 *$\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the unique maximal and minimal PIE in the category of the backtrack* Undo *of $\mathcal{P}$.*

Proof: For Theorem 7.1 $\mathcal{P}^{max}$ is maximal; for Theorem 7.2 $\mathcal{P}^{min}$ is minimal; for Theorem 7.5 $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are respectively the initial and terminal element in the category of the backtrack *Undo* of the same original PIE $\mathcal{P}$. By definition of initial and terminal element, they are unique in the category.

$\square\square$

In Theroem 7.1 and Theorem 7.2, we established that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are a kind of upper and lower bound for any backtrack *Undo* $\mathcal{P}$, in the sense that homomorphisms exist to and from the sets of reachable effects. Moreover, Theorem 7.5 shows that $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are the unique initial and terminal PIE in the category of the backtrack *Undo* of the same original PIE $\mathcal{P}$, so they are in fact the greatest lower bound and lowest upper bound. This means that for any PIE $\mathcal{P}^b$ backtrack *Undo* of $\mathcal{P}$, we cannot have more information than $\mathcal{P}^{max}$ nor less than $\mathcal{P}^{min}$.

7.7 Concluding discussion

At the end of the last Section, we said that for any PIE $\mathcal{P}^b$ backtrack *Undo* of $\mathcal{P}$, we cannot have more information than $\mathcal{P}^{max}$ nor less than $\mathcal{P}^{min}$. But we have not yet explained the meaning of these theorems within specific applications software. To do this, we are going to explain, also with examples, the main characteristic of $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$.

In $\mathcal{P}^{min}$, the set of states is composed by sequences of states, that is to any history provided as input a sequence of states corresponds. All the reached states are stored, besides the repetition of the initial state at the beginning of the sequence. It would be useless to store also such a repetition, because, by performing *Undo* at the initial state, the last would not change. This kind of system allows the user to only perceive the change of state, but he has no information on *how* such a state has been reached. The *Undo* function in minimal systems is simply given by a button or a menu item called *Undo*, but no information on the past actions is given.

Conversely, in $\mathcal{P}^{max}$, all the action history is stored. At any state, the user can have information on all the performed action. The most common representation of such information is by a list.

If we look at real systems they often do not follow an undo policy consistently throughout. However, given these limits, we can see systems which examplify minimal or maximal undo behaviour:

An example of maximal system is given by *Word 6*. As we discussed in Chapter 5, we can consider its undo modes, (phases of using undos and redos), as a single *undo*(n) command once the undo mode is complete. At this level of abstraction the *Undo* mechanism is a pure backtrack *Undo*. If we the consider only the list of actions linked to the *Undo* icon, we have a maximal representation of the backtrack *Undo*.

Although we know of no example of a minimal backtrack undo *editor* (perhaps because it would be too confusing to use), Emacs exemplifies this style of system. In Emacs there is a menu item *Undo* or *Undo more*, but there is no visualisation of the past actions or the position in the history. Actually, Emacs has and *Undo* mechanism which is neither a pure backtrack nor a flip *Undo*. In fact, any time the user performs *Undo*, the reached state is not exactly the previous one but an its copy, so the equivalence is not strong. Moreover, the behaviour is not as the flip *Undo*, since the *Undo* of the *Undo* is not used as the *Redo* function. The behaviour of the *Undo* mechanism in Emacs is a kind of backtrack, but not the pure one. However, its representation is minimal, whithout any information on the past states, so that the user can get lost while interacting.

From the above analysis, the meaning of the theorems of homorphisms is obvious: interacting with maximal systems, any user could have more information on the system state, and could feel more in control of his dialogue. For this reason, the characteristic of maximality is very important, in order to increase the usability of an application software. During the developing phase of an application software, usually the designer chooses the kind

of *Undo* that has to be implemented. Such a choice is done as a balance among the aim of the application, the user needs, the implementing effort, the cost (in sense of the necessary memory), etc. However, before choosing the the kind of *Undo*, the designer cannot leave out of consideration the internal structure of the *Undo* mechanism. By applying formal approaches, behaviour and properties of different *Undo* mechanism are described and the designers should use such information in order to choose the most suitable *Undo* mechanism, so allowing an easy and fruitful interaction.

Conclusions

Many users when interacting with computers make mistakes which must be menaged and, possibly, eliminated. For this purpose, within HCI, recovery functions have been introduced. One of such function is *Undo*.

In this thesis, the *Undo* scope has been analysed. Such an analysis has been performed at three levels, first by describing different recovery functions; then by exploring the kinds of *Undo* mechanisms; finally, by formally characterising the behaviour of a class of *Undo* mechanism, the backtrack *Undo*.

In the first case, depending on the interaction level, we subdivided all the recovery function in three classes: the ordinary recovery functions, available if using reactive systems; the implicit undo, available if interacting with the operating system; and the explicit undo (*Undo*), available when interacting with an application software. Actually, the three classes do not represent a partition of recovery functions; in fact, at the application level, a user can employ both explicit and implicit undo, while at the file level a user can employ implicit undo or any ordinary recovery function. Moreover, it is also possible to perform any ordinary recovery function at the application level, since, by applying some system functions, the working environment may be changed, moving from the application level to the file one, in which any ordinary recovery function may be used. This fact suggested the following hierarchy: $RS \subset HOSI \subset HAI$, where RS indicates reactive systems, $HOSI$ indicates human-operating system interaction, while HAI indicates human-application interaction. The inclusions are based on the availability of recovery functions: the inner set (HAI) is the "richest", the outer one (RS) is the "poorest". In fact, in HAI any kind of recovery function may be applied, explicit undo, implicit undo and ordinary recovery functions, while in RS only ordinary recovery functions are available.

A breadth analysis of the *Undo* scope showed that there is more than one kind of *Undo*. Basically, we can distinguish all the *Undo* mechanisms into

two classes, one in which *Undo* is self-applicable, i.e. the effect of an *Undo* may be deleted by applying another *Undo*, and the other one in which *Undo* is not self-applicable, so that *Undo* of *Undo* may be used as a backtrack tool.

A taxonomy of different kinds of *Undo* has been proposed. Such a taxonomy considers the repetition of *Undo* (if *Undo* of *Undo* is allowed or not) and its granularity (if an *Undo* cancels only one action or a block of actions). A similar analysis has also been done adding the *Redo* function, and a taxonomy of *Undo* - *Redo* mechanism has been proposed.

The choice of the kind of *Undo* to develop a particular application is usually done by the designer which keeps a balance on the aim of the application, the user needs, the implementing effort, the cost (in sense of the necessary memory), etc. However, before choosing the the kind of *Undo*, the designer cannot leave out the internal structure of the *Undo* mechanism. Among the different *Undo* mechanisms we chose the backtrack *Undo* and deep analysis on it has been developed. A formal method, the PIE model, has been applied, in order to describe some properties of such a class of systems.

After giving the definition of conservative encapsulation (which expresses the fact that, given a PIE $\mathcal{P}$ without *Undo* and its augmented system enriched with *Undo* , the system without *Undo* is still inside the augmented one), the definition of behavioural equivalence has been given and we proved that all the systems which are backtrack *Undo* of the same original PIE $\mathcal{P}$ are behaviourally equivalent. This means that, from the user's point of view, they have the same behaviour.

Since such an equivalence is in terms of the effective history, we do not have much information on the sets of states. However, in some sense, we can try to limit the set of states of any backtrack *Undo*, finding a suitable upper bound and lower bound. To this aim, we introduced two PIEs, $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$, which are the backtrack *Undo* of the same original PIE $\mathcal{P}$, with the "biggest" and "smallest" set of states, respectively.

We proved that for any backtrack *Undo* $\mathcal{P}^b$ of the same PIE $\mathcal{P}$, there exists a homomorphism from the set of states of $\mathcal{P}^{max}$ to the one of $\mathcal{P}^b$ and, similarly, there exists a homomorphism from the set of states of $\mathcal{P}^b$ to the one of $\mathcal{P}^{min}$. Moreover, we proved that the class of all the backtrack *Undo* of the same original system $\mathcal{P}$ is a category, whose initial and terminal elements are exactly $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$. Since $\mathcal{P}^{max}$ and $\mathcal{P}^{min}$ are initial and terminal objects in the category, then the two above mentioned homomorphisms are 0-morphisms between $\mathcal{P}^{max}$ and $\mathcal{P}^b$, and between $\mathcal{P}^b$ and $\mathcal{P}^{min}$, and also that such 0- morphisms are unique. The consequence is that $\mathcal{P}^{max}$ and

$\mathcal{P}^{min}$ are the lowest upper bound and the greatest lower bound respectively.

The meaning of this theorem is that we cannot have more "information" than $\mathcal{P}^{max}$ neither less than $\mathcal{P}^{min}$. The maximal system allows the user to have as most information as possible on the system state, so reducing non-determinism and allowing him to feel more in control of his dialogue. For this reason, during the development of an application software, the designers should also take into account the properties of interactive systems described by using some formal methods. In this way, the choice of a suitable *Undo* mechanism can increase the usability of the application software, allowing an easy and fruitful interaction.

An obvious extension of this work is to formally describe the behaviour of the flip *Undo* (for which the *Undo* of the *Undo* is the *Redo* function) and also to prove for this class of systems the theorems of homomorphisms.

Moreover, an analysis of the algebraic properties for systems with single/multi step *Undo* / *Redo* may be done.

Considering the similarity between undo and history mechanisms, backtrack *Undo* properties are also useful to analyse browsing for different interaction histories.

A successive step may also be to enlarge this discussion to the case of the multi-user environments. Some work on the formalisation of *Undo* in collaborative work has already been done. However, it would be better to express properties of such systems with the same formal model. In this way a comprehensive view of interactive systems from the point of view of their undo mechanisms may be obtained.

Bibliography

[1] G.D. Abowd. *Formal Aspects of Human-Computer Interaction.* Technical monograph PRG-97, Department of Computer Science, University of Oxford, 1991.

[2] G.D. Abowd and R. Beale. Users, systems and interfaces: A unifying framework for interaction. In D. Diaper and N. Hammond, editors, *Proc. of HCI'91, People and Computers VI*, pages 73–87. Cambridge University Press, 1991.

[3] G.D. Abowd and A.J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[4] J.E. Archer, R. Conway, and F.B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, (6):1–19, 1984.

[5] A.N. Badre. Methodological Issues for Interface Design: a User-Centered Approach. Research Report DI/DS - 93/01, University of Rome, 'La Sapienza', 1993.

[6] R.W. Bailey. *Human Performance Engineering: a Guide for System Designers.* Prentice Hall, 1982.

[7] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall, 1995.

[8] J.D. Barrow. *The world within the world.* Oxford University press, 1988.

[9] R. Bastide and P. Palanque. Petri Nets with objects for the design, validation and prototyping of user-driven interfaces. In *Proceedings INTERACT'90*, pages 625–631. Elsevier Science Publisher B.V., 1990.

[10] R. Bastide and P. Palanque. Petri Net based design of User-driven Interfaces using the Interactive Cooperative Objects Formalism. In F. Paternó, editor, *Interactive Systems: Design, Specification and Verification*, pages 383–400. Springer Verlag, 1994.

[11] C. Batini, Cerry, and Navathe. *Conceptual Database Design.* Benjamin Cunnings, 1992.

[12] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Human Computer Interaction*, (1):269–294, 1994.

[13] S. Bovair, D.E. Kieras, and P.G. Polson. The acquisition and performance of text-editing skill: a cognitive complexity analysis. *Human-Computer Interaction*, 5(1):1–48, 1990.

[14] D.P. Bovet and P. Crescenzi. *Teoria della complessità computazionale.* Franco Angeli, 1991.

[15] P. Brun and M. Beaudouin-Lafon. A Taxonomy and Evaluation of Formalisms for the Specification of Interactive Systems. In G. Cockton, S.W. Draper, and G.R.S. Weir, editors, *Proc. of HCI'94, People and Computers IX*, pages 197–212. Cambridge University Press, 1994.

[16] V. Bush. As we may think. *The Atlantic Monthly*, 176(July):101–108, 1945.

[17] S.K. Card, T.P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum, 1983.

[18] J.K. Carroll, R.L. Mack, and W.A. Kellogg. Interface Metaphors and User Interface Design. In *Handbook of Human-Computer Interaction*, pages 67–85. North-Holland, 1988.

[19] S.K. Chang, M.F. Costabile, and S. Levialdi. Modeling Users in an Adaptative Visual Interface for Database Systems. *Journal of Visual Languages and Computing*, (4):143–159, 1993.

[20] S.K. Chang, M.F Costabile, and S. Levialdi. Reality Bites - Progressive Querying and Result Visualization in logical and VR Spaces. In *IEEE Symposium on Visual Languages*, pages 100–109. IEEE Computer Society Press, 1994.

[21] I. Cole, M.W Lansdale, and B. Christie. Dialogue design guidelines. In *Human Factors of Information Technology in the Office*, pages 212–241. Wiley, 1985.

[22] M.F. Costabile and R. Mancini. A Methodological Framework to evaluate the VENUS System in a Real environment. VENUS Esprit Project 6398, External Deliverable DI-10/08-010, Gesi, Gestione Sistemi per l'Informatica, 1994.

[23] J. Coutaz. PAC, an object oriented model for dialogue design. In H.J. Bullinger and B. Shackel, editors, *Human-Computer Interaction, INTERACT'87*, pages 431–436. Elsevier Science Publisher B.V., 1987.

[24] A.M. Dearden and M.D. Harrison. Modelling Interaction Properties for Interactive Case Memories. In F. Paternó, editor, *Interactive Systems: Design, Specification and Verification*, pages 301–316. Springer Verlag, 1994.

[25] A. Dix. Formal methods. In *Perspective in HCI -Diverse Approaches*, chapter 2, pages 9–43. ACM Press, 1995.

[26] A. Dix and M. Harrison. Formalising Models of Interaction in the Design of a Display Editor. In H.J. Bullinger and B. Shackel, editors, *Human-Computer Interaction, INTERACT'87*, pages 409–414. Elsevier Science Publisher B.V., 1987.

[27] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *People and Computers: Designing the Interface P Proceedings of HCI'85*, pages 13–22. Cambridge University Press, 1985.

[28] A.J. Dix. *Formal Methods and Interactive Systems: Principles and Practice.* D.phil. thesis, ycst 88/08, Department of Computer Science, University of York, 1987.

[29] A.J. Dix. *Formal Methods for Interactive Systems.* Academic Press, 1991.

[30] A.J. Dix. Que Sera Sera - The Problem of Future Perfect in Open and Cooperative Systems. In G. Cockton, S.W. Draper, and G.R.S. Weir, editors, *Proc. of HCI'94, People and Computers IX*, pages 397–408. Cambridge University Press, 1994.

[31] A.J. Dix. Closing the Loop: Modelling action, perception and information. In *Proc. of the Int. Workshop on Advanced Visual Interfaces AVI'96*, pages 20–28. ACM Press, 1996.

[32] A.J. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction.* Prentice Hall, 1993.

[33] A.J. Dix, J. Finlay, and J. Hassell. Environments for Cooperating Agents: Designing the Interface as a Medium. In *CSCW and Artificial Intelligence*, pages 23–37. Springer Verlag, 1994.

[34] A.J. Dix, R. Mancini, and S. Levialdi. Action, Communication and History. to appear in Proc. of CHI'97, Technical Notes.

[35] A.J. Dix, R. Mancini, and S. Levialdi. Alas I have undone - Reducing the risk of interaction? In *Ancillary Proceedings of HCI'96*, pages 51–56. The British HCI Group, 1996.

[36] D. Duke, G. Faconti, M. Harrison, and F. Paternó. Unifying Views of Interactors. In *Proc. of the Int. Workshop on Advanced Visual Interfaces AVI'96*, pages 143–152. ACM Press, 1996.

[37] A.E. Fischer and F.S. Grodzinsky. *The Anatomy of Programming Languages.* Prentice-Hall, Inc., 1993.

[38] M Green. A survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[39] M. Harrison and A. Dix. A state model of direct manipulation in interactive systems. In *Formal Methods in Human-Computer Interaction*, chapter 5, pages 129–152. Cambridge University Press, 1990.

[40] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction.* Cambridge University Press, 1990.

[41] ISO. *ISO 9126: Software product evaluation - Quality characteristics and guidelines for their use*, 1991.

[42] ISO. *ISO DIS 9241-11: Guidelines for specifying and measuring usability*, 1993.

[43] C. Johnson. Time and the Web: Representing and Reasoning about Temporal Properties of Interaction with Distributed Systems. In

M.A.R. Kirby, A.J. Dix, and J.E. Finlay, editors, *Proc. of HCI'95, People and Computers X*, pages 39–50. Cambridge University Press, 1995.

[44] M.W. Lansdale and T.C. Ormerod. *Understanding Interfaces.* Academic Press, 1994.

[45] G.B. Leeman. A Formal Approach to Undo Operations in Programming Languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87, 1986.

[46] S. Levialdi, P. Mussio, M. Protti, and L. Tosoni. Reflection on Icons. In *IEEE Symposium on Visual Languages*, pages 249–254. IEEE Computer Society Press, 1993.

[47] H.R. Lewis and C.H. Papadimitriou. *Elements of theory of computation.* Prentice-Hall, 1981.

[48] J.C.R. Licklider. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics HFE*, 1(1):4–11, 1960.

[49] P. Linz. *An Introduction to Formal Languages and Automata.* D.C. Heath and Company, 1990.

[50] L. Macaulay. *Human-Computer Interaction for Software Designers.* International Thompson Computer Press, 1995.

[51] M. Macleod. An introduction to Usability Evaluation. NPL Report DICT 102/92, National Physical Laboratory, UK, 1971.

[52] M. Macleod. An Introduction to Usability Evaluation. NPL Report DICT 102/92, National Physical Laboratory, UK, 1992.

[53] F. Maddix. *Human-Computer Interaction, Theory and Practice.* Ellis Horwood Limited, 1990.

[54] J. Maissel, M. Macleod, A. Dillon, C. Thomas, R. Rengger, M. Maguire, M. Sweeney, and R. Corocran. *Context guidelines handbooks, Version 2.1*, 1993.

[55] R. Mancini. Interacting with a Visual Editor. In *Proc. of the Int. Workshop on Advanced Visual Interfaces AVI'96*, pages 125–131. ACM Press, 1996.

[56] R. Mancini, A. Dix, and S. Levialdi. Formal and Informal Definitions of Undo. Research Report RR9611, School of Computing and Mathematics, University of Huddersfield, UK, 1996.

[57] M. De Marsico and R. Mancini. Usability through Iconic Interfaces. In *Proc. of OZCHI'93*, pages 264–266. CHI Special interest Group of the Ergonomics Society of Australia, 1993.

[58] R.B. Miller. Human ease of use criteria and their trade-offs. IBM Report TR00.2185, Poughkeepsie, NY:IBM Corporation, 1971.

[59] R. Milner. *Communication and concurrency.* Prentice Hall, 1988.

[60] R.N. Moll, M.A. Arbib, and A.J. Kfoury. *An Introduction to Formal Language Theory.* Springer-Verlag, 1987.

[61] S.J. Mountford. What can Users tell about User Interface? In *Proc. of the Int. Workshop on Advanced Visual Interfaces AVI'92*, pages 103–107. World Scientific Press, Singapore, 1992.

[62] B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *Proceedings of CHI 96, Vancouver, BC, Canada*, pages 260–267. ACM Press, 1996.

[63] J. Nielsen, editor. *Usability Engineering.* AP Professional, 1993.

[64] D.A. Norman. Categorisation of action slips. *Psychological Review*, (88):1–15, 1981.

[65] D.A. Norman. Cognitive Engineering. In D.A. Norman and S. Draper, editors, *User-Centered System Design*, pages 31–62. Erlbaum, 1986.

[66] A. Monk P. Wright and M. Harrison. State, display an undo: a study of consistency in display based interaction. Technical report, University of York, 1992.

[67] P. Palanque and R. Bastide, editors. *Proc. of the Eurographics Workshop on Design, Specification and Verification of Interactive Systems '95.* Springer Verlag, 1995.

[68] F. Paternò, editor. *Interactive Systems: Design, Specification and Verification.* Springer Verlag, 1994.

[69] F. Paternò. Formal Methods for Multimodal Interactive Systems. In *Tutorial 11, HCI'96*. The British HCI Group, 1996.

[70] J. Pearsall and B. Trumble, editors. *The Oxford English Reference Dictionary*. Oxford University Press, 1995.

[71] G.E. Pfaff. *User Interface Management System*. Springer Verlag, 1985.

[72] A. Prakash and M. J. Knister. Undoing Actions in Collaborative Work. In *CSCW'92 Sharing Perspectives, Proc. of the Conference on Computer-Supported Collaborative Work*, pages 273–280. ACM Press, 1992.

[73] J. Preece and L. Keller, editors. *Human-Computer Interaction, Selected Readings*. Prentice Hall, 1990.

[74] T.V. Raman and D. Gries. Interactive Audio Documents. *Journal of Visual Languages and Computing*, (7):97–108, 1996.

[75] C. Rouff. Formal Specification of User Interfaces. *ACM SIGCHI Bulletin*, 28(3):27–33, 1996.

[76] C. Runciman. From abstract models to functional prototypes. In *Formal Methods in Human-Computer Interaction*, chapter 7, pages 201–232. Cambridge University Press, 1990.

[77] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory*. Prentice Hall, 1988.

[78] A. Salomaa. *Formal Languages*. Academic Press, 1973.

[79] F. Schile and T. Green. HCI formalisms and cognitive psychology: the case of task-action grammars. In *Formal Methods in Human-Computer Interaction*, chapter 2, pages 9–62. Cambridge University Press, 1990.

[80] B. Shackel and Richardson, editors. *Human Factors for Informatics Usability*. Cambridge University Press, 1991.

[81] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3):237–256, 1982.

[82] A. Silberschatz and P. Galvin. *Operating Systems Concepts -Fourth Edition*. Addison-Wesley, 1994.

[83] R. M. Stallman. EMACS: The extensible, customizable self- documenting display editor. *ACM SIGPLAN Notices*, 16(6):147–156, 1981.

[84] H.W. Thimbleby. *User Interface Design.* Addison Wesley, 1990.

[85] S. Treu. *User Interface Design - A Structured Approach.* Plenum Press, 1994.

[86] J.S. Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, 1984.

[87] P. Wegner. Tradeoffs between Reasoning and Modeling. In *Research Directions in Concurrent Object-Oriented Programming*, chapter 2, pages 22–41. MIT Press, 1993.

[88] Y. Yang. Undo support models. *International Journal of Man-Machine Studies*, (28):457–481, 1988.

[89] R.M. Young and G.D. Abowd. Multi-Perspective Modelling of Interface Design Issues: Undo in Collaborative Editor. In G. Cockton, S.W. Draper, and G.R.S. Weir, editors, *Proc. of HCI'94, People and Computers IX*, pages 249–260. Cambridge University Press, 1994.