PrePrint

Upside down ∀s and algorithms computational formalisms and theory

Alan Dix

Chapter 14 in
HCI Models, Theories, and Frameworks: Toward a Mulitdisciplinary Science.
John Carroll (ed.)

San Francisco, USA: Morgan Kaufmann, 2003.
ISBN 1-55860-808-7. pp. 381-429.

http://www.hcibook.com/alan/papers/theory-formal-2003

14 CHAPTER Upside-Down ∀s and Algorithms—Computational Formalisms and Theory

Alan Dix Lancaster University, England

14.1 MOTIVATION

The time delay as Internet signals cross the Atlantic is about 70 milliseconds, about the same time it takes for a nerve impulse to run from your finger to your brain. Parallels between computation and cognition run as far back as computers themselves. Although at first it feels as if the cold, abstruse, more formal aspects of computation are divorced from the rich ecology of the human-computer interface, the two are intimately bound. Mathematics has also been part of this picture. Indeed, the theory of computation predates digital computers themselves, as mathematicians pondered the limits of human reasoning and computation.

There are a number of aspects of this interplay between computation, mathematics, and the human-computer interface.

First, understanding your raw material is essential in all design. Part of the material of human-computer interaction (HCI) is the computer itself. Theoretical and formal aspects of computing can help us understand the practical and theoretical limits of computer systems, and we can thus design around these limits.

Second, diagrams, drawings, and models are an integral part of the design process. Formal notations can help us sketch the details of interaction, not just the surface appearance, of an interactive system, and we can thus analyze and understand its properties before it is built. This is the area that is typically called "formal methods" within HCI, and we'll look at an example of this in Section 14.2.

Third, various techniques from mathematics—simple counting to sophisticated equations—may be used to reason about specific problems in HCI. In this book, we see chapters including Fitts' Law, a logarithmic regression; information foraging theory, which involves differential equations; not to mention the heavy reliance on statistical modeling and analysis of virtually all quantitative empirical work.

Finally, the design artifact of HCI involves people and computers working together within a sociotechnical whole. Among the many political, social, and emotional aspects of this interaction, there is also an overall computational facet. The theory of computation has, from the beginning, spanned more than mere mechanical computation, and conversely an understanding of digital computation can help us understand some of the complexity within rich organizational ecologies.

14.1.1 What Is Formal?

As with all words, "formal" is used to mean different things by different people and in different disciplines. In day-to-day life, formal may mean wearing a dinner jacket and bow tie or using proper language. That is, formal is about the outward form of things—a formal greeting may belie many emotions beneath the surface.

Taken strongly, formalism in mathematics and computing is about being able to represent things in such a way that the representation can be analyzed and manipulated without regard to the meaning. This is because the representation is chosen to encapsulate faithfully the significant features of the meaning.

Let's see what this means in simple mathematics. One night you count the cockroaches on the wall—213. The next night you count again—279. Now because 279 is bigger than 213, you can assert that there were more cockroaches on the second night than the first. How do you know? Did you line up the cockroaches in queues? No! You know because you compared the numbers. The numbers 213 and 279 represented faithfully the significant feature of the cockroaches. Of course, the cockroaches on the first night might have been bigger, a different color, more friendly, so the numbers don't capture everything.

Even looking at something as simple as cockroaches, we can see both the power and limitation of formalism. It allows us to discuss certain features succinctly and with precision, but without the things they represent being present. However, by its nature it excludes features that are not represented

When we write, when we speak, when we draw, symbols stand for things that are not present. In interfaces, a drawing of a potential screen design allows us to discuss that design even though it is not there. However, the drawing does not allow us to discuss the dynamics (perhaps a storyboard might allow this for a single scenario) or the effects of actions on underlying data, on the world, or on other users. Certain formal descriptions of a system allow this. Think again of the cockroaches. If we were comparing cockroaches in two different rooms, it would, in theory, be possible to collect them in jam jars and then line the cockroaches up to see which room had more. However, if we had collected the first night's cockroaches in a jam jar, we would not have had 279 cockroaches the second night to collect as some would be stuck in the jar. The formalism of counting not only makes the comparison easier, it makes it possible.

Formal descriptions often include place holders for unspecified things, for example, a furniture designer may draw a homunculus with limbs in the right proportions. The homunculus stands for *anyone* of the relevant height, arm length and so on. Notice how powerful this is—not only can we talk about a specific person who isn't present, but we can talk about anyone at all with particular properties. This is used to particular effect in mathematics where we can say things like: "for any number n, $n+1$ is greater than n." Here the place holder "n" stands for any number whatsoever. This power to talk about an infinite amount of things (every number!) in a finite expression is one of the things that makes mathematics so powerful.

Because the real thing does not need to be present, it may be something abstract, including something like a sequence of interactions. A storyboard would allow us to talk about one example of a fixed sequence of interactions, whereas in a formal system we can usually talk about things we would like to be true of any interaction sequence. And, more abstract again, we can talk about any system of a particular kind and all the possible interactions a user could have with that system For example, Mancini and I worked on issues surrounding **Undo** and were able to show that any system with particular kinds of **Undo** would need to be implemented in effectively the same way (Dix & Mancini, 1997; Dix, Mancini, & Levialdi, 1997).

Finally formalisms force you to think. It would be easy to look at the wall one night and think "there are more cockroaches than yesterday." Perhaps the wall just looks more densely covered. However, when we start to count the cockroaches, we need to be a lot more clear. What counts as a cockroach—should I include that mess where I hit a cockroach with my shoe? The cockroaches keep running in and out of cracks—do I mean every cockroach including those hidden in the cracks (perhaps I could put spots of paint on them), or do I just mean every cockroach that is out at a particular time, say 11 P.M.?

when you count cockroaches you have to decide what counts as a cockroach

Sometimes forcing things into conceptual boxes is unproductive (think how many hours have been wasted arguing whether a platypus is a bird or a mammal), but often the very act of being precise makes us think more clearly about problems. While using formal methods to model various aspects of interaction, I have often found that it is the process of creating the model that is more important than anything I do with it.

14.1.2 The Myth of Informality

There is a fashionable tendency to decry the formal, mathematical, or precise in favor of softer, more informal techniques. This is always a popular stance, perhaps because of the rise of post-modernism, perhaps because people feel powerless, like cogs in the unbending bureaucracies of modern society. Seminal books such as Suchman's *Plans and Situated Actions* (1987) and Winograd and Flores' *Understanding Computers and Cognition* (1986) have been successful in part because they appear to present atheoretic, highly contextual, and aformal approaches. (In fact, both have strong theoretical stances of their own and were written in the context of an overformal design environment at the time.)

As already noted, there are many other disciplines and many aspects of HCI where this is the right approach, and attempts to formalize such areas are misguided. However, maintaining the opposite view, the myth of informality, and eschewing formalism completely is both illusory and ultimately dangerous.

The design process may well be informal, but the designed product is anything but. A computer is the most formal artifact possible; *all* it does is interpret meaningless symbols according to formal rules.

Maintaining the myth of informality means that the effects of the intrinsic formality of a computer system have not been considered during design. Would you buy a ship from an engineer who lived in the Sahara?

There are arguments for and against formalizing certain phenomena, but not computer systems. Any design process that produces computer systems as part of its final product (albeit within a rich sociotechnical whole) is dealing with formality.

The choice is simple: Do we deal with the formality in knowledge or in ignorance?

14.1.3 Chapter Structure

The next section will examine some simple examples in order to demonstrate the utility of this approach. We will then look at some of the history of the use of formalism in mathematics and computing from Aristotle to Alan Turing. This will then lead to a more detailed review of the different ways in which formal methods and computational theory have been used in HCI. In order to validate this, a more substantial case study will show how adoption of a simple formal notation for dialogue design led to a 10-fold improvement in productivity and more robust systems. The final section of the chapter will establish the current state of the field and will look forward to the future potential of these techniques, especially in emerging application areas such as ubiquitous computing.

14.2 OVERVIEW OF ISSUES AND FIRST STEPS IN FORMALISM

In order to show some of the aspects of formal methods in interaction design, we'll look at the way simple diagrammatic formalisms can help us examine two simple devices, a wristwatch and an alarm panel. From these, we will see some important general lessons about the advantages of using more formal techniques.

14.2.1 Two Examples

Some years ago, my daughter had a digital wristwatch. The watch had two buttons and came with a set of closely written instructions covering a slip of paper about 2 inches wide (5cm) and 15 inches long (40cm). Figure 14.1 shows one diagram on the paper (much larger than life size!). The two buttons were labeled A and B; this diagram was intended to show the main functions of button A.

From this diagram, we can see that button A switches the watch between four main modes:

Time display: normal mode showing current time and day

Stop watch: timing to seconds and hundredths of a second

Time setting: to set/update the time

Alarm setting: to set the built-in alarm

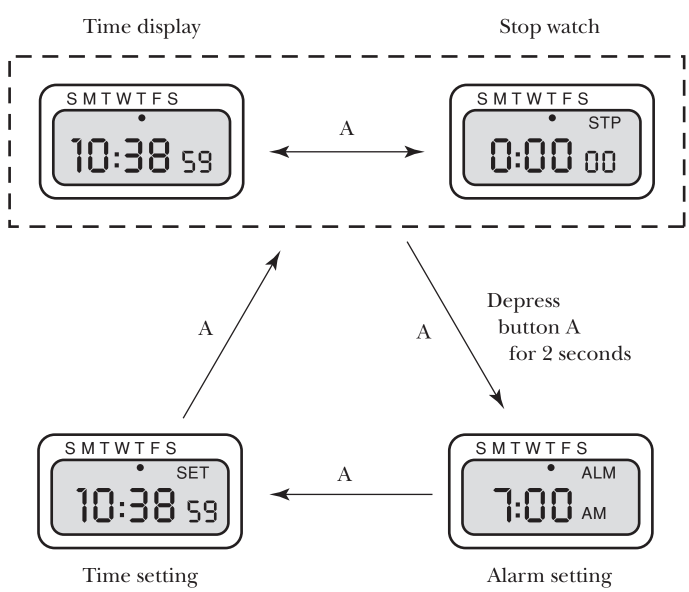
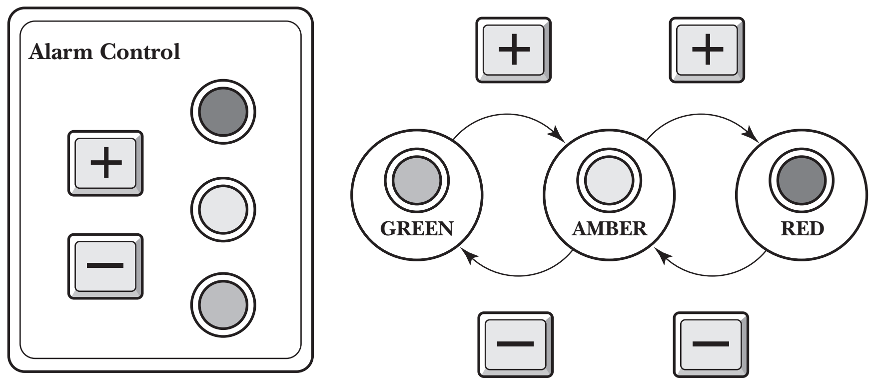


FIGURE 14.1 Digital watch instructions.

This diagram, produced as part of the user documentation, can be seen as a state transition network (STN), a type of formal specification of part of the functions of the watch. STNs have been used extensively in interface specification. They are often written as small circles or boxes representing the potential states of the systems, with arrows between the states showing the possible transitions. The arrows are labeled with the user action that initiated them. Describing states is always difficult, and the digital watch instructions do this by simply drawing what the watch will look like in the state.

Consider an even simpler interface—perhaps the security control panel at a top secret establishment, shown in Figure 14.2 (a) and Plate 13 of the color insert. It has two buttons labeled "+" and "−"—these control the current alarm state, which can be at three levels, or three lights, green, amber, or red.

Imagine we have been given a state transition network for this panel; see Figure 14.2 (b). Just looking at the STN—without knowing what it is about—we can ask some questions: What does the "+" button do when the state is red; what does the "−" button do when the state is green? This is a formal analysis—we don't need to know what the STN means, but we can see just from the pattern that something is missing. In many applications, such as a remote control for a television, we would probably want the "+" button to cycle back round from red



(a) control panel (b) state transition network

FIGURE 14.2 Top secret control panel. (See Plate 13 of the color insert for a color version of this figure).

to green, but in this application—an operator under stress perhaps—we would probably not want to risk accidentally changing from red alert to green. The formal analysis tells us *that* something extra is needed; the contextual understanding tells us *what* it is.

Looking again at the digital watch, there is a similar story. At first it looks as though it is complete. Only one button is being considered, and the diagram shows what it does in every state. Or does it? What happens from the Time setting state—does it go to Time display, Stop watch, or perhaps whatever state it was in before going into Alarm setting? In fact, after a quick bit of experimentation, it turned out that it always goes to the Time display state.

Also, look at the transition labelled "Depress button A for 2 seconds." This suggests two things:

- Time is important.
- We need to think about depressing and releasing the button as separate events.

This immediately prompts a series of questions:

- Does time matter in either of the "setting" states?
- Do the transitions labeled simply "A" mean :when A is depressed" or "when A is released"?

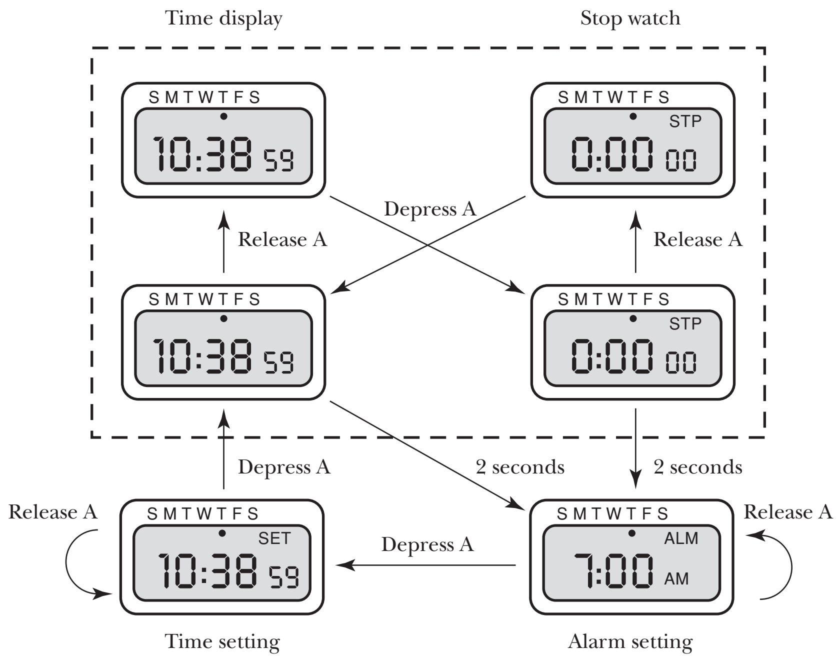


FIGURE 14.3 Digital watch states in detail.

Further experiments showed that the transitions between “Time display” and “Stop watch” happened when the button was pressed. However, clearly the system “remembered” that the button was still down in order to change to “Alarm setting” state after two seconds. Figure 14.3 shows the full picture with the “Time display” and “Stop watch” states each split in two to represent this “remembering.”

14.2.2 Lessons

These examples demonstrate several key features of formal descriptions.

formal analysis. It is possible to ask questions about the system from the formal description alone. For example, in the alarm system, we noticed that “+” was missing from the red state. We didn’t need to know anything about what the diagram meant (nuclear meltdown, or simply color chooser); we just used the *form* of the diagram alone.

S
R
L

early analysis. In the case of the watch we actually had the artifact; because we could look at the diagram, however, we could have performed the same analysis before a prototype was available. Even with rapid prototyping, this can save precious resources.

lack of bias. If we had been testing the alarm system without the formal analysis, we might not have thought to try out "unusual" actions like pressing "+" when the alarm was already in "red" state. Testing is often biased toward "normal" behavior—what the designers expect the users to do. Formal analysis helps to break this bias.

alternative perspective. In a single diagram, the formal description looks at the system with every different potential user input. Drawings of the watch early in design would show what it was like (but in greater detail) at individual moments of time. A fully working prototype would give a better feeling for what interaction with the watch was really like, but it still does not let you see different possibilities except by repeatedly experimenting. Different representations therefore allow us to see different things about a design. Furthermore, different kinds of formal representation can allow yet more views on the artifact during design.

forcing design decisions. The watch example is taken from the user documentation and so would not be expected to show all possible fine behaviors (it would simply be confusing). However, one wonders whether the detailed behavior in Figure 14.3 was the result of the designer thinking about what was wanted or arbitrary decisions made while coding the internals of the watch. Using formal representations, the designer is forced to make these user-interface decisions explicitly and to communicate those decisions to the implementor.

On the other hand, the example in Figure 14.3 shows only the behavior of one of the two buttons of a pretty simple device. Formal descriptions, by making you be precise, can become complex through sheer level of detail. However, this extra detail is not being created for the sake of the formal description but is simply exposing the extra decisions that will be there in the final artifact. If they aren't there in the design documents, they will be in the code!

14.3 SCIENTIFIC FOUNDATIONS

In this section we'll look at the history of formalism and how this feeds into current uses of formal notations, formal methods, and general computational theory. Much of this story is about the gradual discovery of the fundamental limits of human knowledge in general and computation in particular.

14.3.1 A Brief History of Formalism

The roots of formalism go back a long way, certainly to Euclid's axiomatization of geometry and to Aristotle and the formalization of logical inference (Rouse Ball, 1908). It is Euclid who had the most significant impact on mathematical education. It is amazing to think that Euclid's *Elements* was still being used as a high school text book 2200 years after it was written! However, it is Aristotle who is the more "modern" thinker. Despite coming through the Platonic school of perfect forms, he was able to look to the world as it is—the beginnings of empiricism. But he also based this on two pillars of self-consistent and sufficient basic truths in theology and logic (or mathematics).

Of course, many things happened over the next 2,000 years: Arabic mathematics with the introduction of zero and the modern digits, and the development of calculus (differentiation and integration), which enabled the formulation of modern physics, engineering, economics, and much more. However, this is not the story of mathematics, but formalism, so we'll skip forward to the nineteenth century.

Galois is one of my heroes.[1] He set "group theory" on a formal footing. Group theory is, in one sense, a very abstract branch of mathematics; yet, like numbers, it is an abstraction of common phenomena, such as the number of ways you can put a computer back into its box (upside down, rotated 180 degrees, etc.). Group theory can tell us that the slider puzzle shown here cannot be solved, and it also underlies modern quantum mechanics, but above all Galois set it in a formal framework that is the pattern for much of modern mathematics. That is why he is an important step in the story of formalism, but not why he is my hero. Galois' theory did all this and also was the key to solving a whole range of open problems about the roots of polynomials and what can be constructed with ruler and compasses, many of which were unsolved since the height of Greek geometry 2,000 years previously. I particularly like the fact that his theoretical framework largely proves what you can't do, foreshadowing some of the great results of the twentieth century. And he did all this before, at the age of 20, he was killed in a duel. Beat that!

The nineteenth century saw Cantor's formulation of set theory, which is at the heart of virtually all mathematics and is also the foundation of many "formal"

---

[1] Evariste Galois, born in Paris in 1811.

methods in computing.[2] For example, it is set theory that gives us the vocabulary to talk about the set of possible keys the user could press:

```
Input = { 'a', 'b', . . . , 'z', '0', '1', . . . }
```

From this and functions between sets, we can talk about the set of states of a system, and the function that maps the current state to the new state depending on the current input:

```
update: State ∞ Input ∅ State
```

Although there were problems and setbacks, the nineteenth century in mathematics, as in so many fields, was one of rising optimism. At the turn of the century, mathematicians' success was so great that they began to ask fundamental questions of the discipline itself: Could all mathematics be performed strictly formally, without insight or understanding, simply by playing out the rules? Various major steps were taken, and problems in the theory of infinite sets were solved (problems uncovered by Bertrand Russell's paradox of the set of sets that don't contain themselves). Whitehead and Russell began a monumental work of proving basic mathematics with total rigor from logical foundations. It took a major volume to even get as far as statements such as 1+2=2+1, but progress was being made, and the pair worked their way through several more tortuous volumes.

Not only was mathematics itself progressing, but throughout the nineteenth century mathematics was being used as the language of creation: formulating and integrating the basics of physics, especially in Maxwell's equations, which brought magnetism, static electricity, and electrical current within a single coherent mathematical theory. It seemed as though there would be no end to this progress; the complete and final formulation of all things was within human grasp.

Remember Aristotle's twin pillars: theology and logic. Towards the end of the nineteenth century, Nietzsche, reflecting this spirit of humanistic optimism and self-sufficiency, declared the "death of God," and in the process discarded one of Aristotle's pillars.

By the 1920s, however, the foundations of this optimism had been systematically undermined. The discovery of quantum mechanics and relativity showed that the complete, comprehensive treatment of physics was not around the

[2] Georg Cantor (1845–1918), German mathematician and professor at Halle.

corner and that the world did not behave in ways that made sense at all. World War I hardened an already growing postcentennial cynicism and laid to rest, with the corpses in Flanders, the belief in the moral ascent of man.

But Aristotle's second pillar stood firm and, in the cloistered halls of Cambridge University, Whitehead and Russell labored on. Then, in an obscure conference hall, a young mathematician, Gödel, stood up and announced the death of mathematics. The last pillar had fallen.

14.3.2 The Limits of Knowledge

I said that Galois, in proving the impossibility of many problems, also prefigured the twentieth century. The nineteenth century appeared to be an unstoppable chain of achievement, leading to the belief that everything was possible. In contrast, the successes of the twentieth century have been in understanding what is not possible.

Gödel began this process in mathematics. He proved that any formal system was either incomplete (it couldn't prove some things even though they are true) or inconsistent (proves things that are false), and, moreover, we can't tell which of these is true of any particular system except by applying a different system . . . and then we don't know about that one (Kilmister, 1967).

Gödel showed that our knowledge in mathematics is patchy and full of holes, like Swiss cheese. Not accidental holes, because we haven't got round to finding something out yet, but fundamental holes because they can never be filled.

Given his fundamental reshaping of the intellectual landscape, Gödel should surely rank alongside Einstein. The physics of quantum mechanics and relativity can only be understood through their mathematics, but Gödel cast in doubt the very meaning of the mathematics itself. Gödel found his evangelist in Hofstadter, whose cult book Gödel, Escher, Bach: An Eternal Golden Braid popularized many complex issues of incompleteness, self-similarity, and self-reference (Hofstadter, 1979).

14.3.3 The Theory of Computing

The roots of the theory of computation also began during the 1930's, much of it before the first electronic computers. Church and Turing worked on understanding what was possible to compute, whether by clerks in a bank or by a machine, but each dealt with very different models of computation. Church's

Some computer programs finish their job quickly, and others may take a long time but eventually "halt" and give their result. However some programs simply never finish at all; they never halt. Imagine that you have bought a clever program (let's call it **will_halt**) from magisoft.co.uk that tells you whether other programs will eventually halt or not. You then write the following:

```
my_program:
    if will_halt(my_program)
              loop forever
    otherwise stop
```

Does your program halt? If it does, and **will_halt** works this out, then your program will loop for ever; if it doesn't halt and **will_halt** works this out, then it will stop. That is, if it halts it doesn't and if it doesn't it does. Something is wrong! You sensibly return **will_halt** to magisoft as it clearly can't possibly work.

BOX 14.1 The Halting Problem

lambda calculus is a sparse algebraic description of functions using only lines of λs, variables (x,y, . . .), and brackets combined with a small number of "reduction rules." In contrast, the Turing machine consists of a small device running over an infinitely long tape. The magic is that, in a milestone result, it was found that these two models, although different in methods of operation, were exactly equivalent in what they could do. Other models of computation were also found to be equivalent. This is taken as justification (not proof) of the Church–Turing thesis (actually coined by Kleene) and variants, which is a broader hypothesis stating that all computation is fundamentally the same (see the Stanford Encyclopedia of Philosophy).

However, this early theory did more than model computation; it also found its own limits. In a variety of results, most famously the halting problem (see Box 14.1), the theorists of computation found that there were certain useful things that it was impossible to compute.

So, from the very beginning, computation theory has considered its limitations.

14.3.4 Complexity

As important as whether something finishes is how long it takes. If something is going to take 100 years to happen, I don't think I'll wait around. A branch of computing called *complexity theory* deals with these questions "how long," "how much memory," "how many network messages," and, in parallel computers, "how many processors."

Let's think of an electronic newsletter. Each day I email it to my subscribers. If there are 10 subscribers, there are 70 messages per week; if there are 100 subscribers, there are 700 messages per week; for 1000 subscribers, 7000 messages. The number of messages rises linearly with the number of subscribers. If there are n subscribers, this is usually written $O(n)$ (the "O" stands for "order of").

Now let's imagine a mailing list. Each morning, everyone reads all the messages on the list and sends one of their own to *everyone* on the list (including themselves). If there are 10 people, then there will be 700 messages a week—10 people each day sending 10 messages. If there are 100 people, there will be 70,000 messages (7 days × 100 senders × 100 recipients); 1000 people leads to 7 million messages per week. This time, the number of messages rises with the square of the number of people. This is written $O(n^2)$.

Some problems are even worse. Imagine you are a manager trying to work out who should do which job. You decide to try every possible combination and each day assign people accordingly. Say there are two people, Ann and Bob, and two jobs, artist and builder. On day 1 you try Ann as artist and Bob as builder; on day 2 you try Ann as builder and Bob as artist. Two days and you've tried the lot. Now imagine there are three people—Ann, Bob, and Chris—and three jobs—artist, builder, and clerk. This time it is a little more complicated, so you write out the schedule as seen in Table 14.1:

Notice this time that we need six days for three people. When Dave joins the firm, you branch out into medicine and take 24 days to try four people in four jobs. You grow further and find that five people in five jobs takes 120 days, and six people in six jobs takes 720 days. This number is growing as the factorial of the number $O(n!)$. (See Box 14.2)

At this point you enlist Pete (whom you know is a good programmer!) to write a program that, based on data in the personnel database, simply simulates assigning people to roles and works out the best combination. Pete is good. The program takes only a microsecond to simulate each assignment, so you test the software by running it against your existing six people; in 720 microseconds, less than a millisecond, it gives the answer and agrees with the real experiment. Now that you have solved your personnel-management problems you happily grow

	Ann	Bob	Chris
day 1	artist	builder	clerk
day 2	artist	clerk	builder
day 3	builder	artist	clerk
day 4	builder	clerk	artist
day 5	clerk	artist	builder
day 6	clerk	builder	artist

TABLE 14.1 Schedule of all possible combinations of jobs for three people

The factorial of a number is the product of all the numbers less than or equal to it. For example, 6! = 6×5×4×3×2×1. It grows very rapidly, even faster than an exponential.

n	n!	
1	1	
2	2	
3	6	
4	24	
5	120	
6	720	
7	5040	
8	40320	
9	362880	
10	3628800	
11	39.9	million
12	479.0	million
13	6227.0	million
14	87178.3	million
15	1307674.3	million

BOX 14.2 Factorial

S
R
L

further. When you have 10 employees and 10 jobs, the system checks and finds the best combination: in less than four seconds (3.6288 seconds to be exact) it gives the answer. As you grow to 11 and 12 employees, it seems to be a little sluggish responding, but you put this down to having the monthly accounts running at the same time. However, when you hire your thirteenth employee things start to go wrong. You wait and wait, then ring Pete and get his answering machine; just as he rings back, an hour and three quarters later, the computer comes back to life and gives the answer. "Ah well, it's just one of those funny things computers do" you agree. On the fourteenth employee, you set it going in the morning and then go out to client meetings for the rest of the day. When you get in the next morning, the computer screen looks frozen, perhaps it has crashed; you go to get a cup of coffee and it is waiting there with the answer when you get back . . . curious. But with the fifteenth employee, things are clearly not right—the program seems to run and run but never stop. Pete is on holiday for two weeks, and so you have to wait for his return. When he does, you call him, and he says "ah, I'll have to think about it." The next day he calls and says, "Look at the screen now"—and the answer is there. "That's amazing," you say. "How did you do it?" Then he admits that 15 employees simply took 15 days to complete, but that the sixteenth would require 242 days; if you want to grow to 20 employees, you will need to wait seventy-seven thousand years for the answer. Twenty-four employees would take 20 billion years, longer than the age of the universe. Ah, well, they do say small is beautiful.

It is so easy to assume that computers are fast and can do anything, but if the underlying problem or algorithm doesn't scale no amount of speed will be enough. Problems like the last one that involve trying lots of combinations are known as nonpolynomial as there is no polynomial (power of n) that bounds them. Even quadratic, $O(n^2)$, algorithms grow in difficulty pretty rapidly (as in the mailing list), and to be really scalable one tries to find things that aren't too much worse than linear. For example, simple sorting algorithms are $O(n^2)$, but there are faster algorithms that are $O(n \log_2 n)$. This means that sorting 8 items takes about 2×1 steps, 4 items takes 4×2 steps, 8 items 8×3, 256 items 256×8, and so on. This is longer than linear, but massively faster than a slow quadratic sort (32 times faster for 256 items, and 100 times faster for 1024 items).

14.3.5 Good Enough

Notice that the halting problem is about a program that can *always* tell us whether other programs halt; the personnel allocation program tries *every* combination in order to find the *best* solution. This is traditional algorithmics—deterministic methods that, excepting for simulations, dominated the first 25 years of programming and still form the main part of virtually all current systems and computing syllabi.

However, since the mid-1970s, a second strand has emerged that is focused on *usually* getting an answer, or trying *enough* things to get a *reasonably good* answer. Methods in this strand include simulated annealing, neural networks, and genetic algorithms, but there are also numerous more specialized algorithms that employ elements of approximation or randomness to solve problems that would otherwise be impossible or intractable. Many of these techniques are based on analogies with physical or biological systems: simulated annealing—the cooling of metals; neural networks—brains; genetic algorithms—natural selection; and recent work has been inspired by the behavior of colonies such as ants or bees.

14.3.6 Agents and Interaction

The early theory of computing was also very much concerned with all or nothing computation—you give the computer a problem and then wait for the result. In HCI, we are more interested in interaction where we constantly give little bits of input to a device and get little bits of response. To be fair, computational models that involve interacting agents have been around since the early days of computing. For example, cellular automata (CA), made popular by Conway's Game of Life (Berlekamp, Conway, & Guy, 1982; Gardner, 1970)—see Box 14.3—were originally proposed by von Neumann, one of the pioneers of computing (1956, 1966). However, for a long time such interactions were mainly concerned with what happened *within* a computing system (typically as a computational model or parallel or distributed computing). This has gradually changed, and it is now common to consider models of multiple agents interacting with one another, the environment, and human users.

This view of computation as involving interacting entities has both practical and theoretical implications. My own studies of human processes emphasize the importance of initiative—who makes things happen and when (Dix, 1998; Dix, Ramduny-Ellis, Wilkinson, 2002). This is certainly a key element in many areas of practical computing, too, but until recently not a major part of the vocabulary of fundamental computation theory. Wegner has suggested (1997) that the fundamentally different nature of interaction may mean that interacting computers are fundamentally more powerful than all-or-nothing computation.

Imagine a large board of squares such as a chess or Go board. Each square either contains a token (is alive) or doesn't (is dead). Now, for each square, you look at its eight neighbors and apply the following simple rules:

overcrowding: If more than three are alive, the square becomes (or stays) dead.

loneliness: If less than two are alive, the square becomes (or stays) dead.

birth: If exactly three are alive and the square is dead, it becomes alive.

Otherwise, the square stays as it was.

The rules are regarded as operating at the same moment over the whole board (that is, not in sequence).

2	1	3	4
5	6	7	8
9	10		11
13	14	15	12

If you regard each square as a small computational device, Life is a form of cellular automata (CA). CA differ in how many states each device has (just two here, alive or dead), how many neighbors are considered, the geometry of the "board" (2D, 3D, different shaped grids), and the complexity of the rules.

BOX 14.3 Conway's Game of Life

Working at a time when there were no physical computers, Turing was radical in ensuring that the models of computation he used were physically realizable. However, as real computers became available, this view was somehow lost, and computation theory became more and more abstract. And to some extent we now see signs of a more grounded theory as computational devices are scattered among our everyday lives. I call this more physical view *embodied computation* and find that it radically changes the way I personally look at computation (Dix, 2002a).

14.3.7 Notations and Specification

Although a large part of mathematics is about manipulating formulae, it is remarkably laid back about notations. There are some fairly standard symbols (e.g., basic arithmetic, upside down ∀ for "forall" and back to front ∃ for "exists"), but these are regarded as a convention rather than a standard. In individual parts of mathematics and in individual papers and books, variants are used for the same thing and special notation introduced on an *ad hoc* basis.

Computing has always been more conservative about notation. Perhaps this is because of its connections with the more "formal" parts of mathematics, perhaps because of links with electrical engineering where standardization is more important (if a mathematician gets confused, she wastes paper; if an electrician is confused, he dies). Possibly, also, it is because computer science has had to create notations (programming languages) to communicate with computers and has let the same mindset leak into notations to communicate with one another!

No matter the reasons, notation is very important. However, this doesn't mean that there is no *ad hoc* development of notations. Indeed, in some areas of computing—both in programming languages and in formal areas—it seems as if every paper has a new notation. The difference is that the notation itself is seen as significant: the end, not just the tool. The few notations that have become widely used have earned their originators cult status and have found their adherents in often hostile factions. Notation is the religion of computer science!

In mathematics, notations are most widely used to talk about general properties of classes of mathematical structure. In contrast, computing notations are most often developed to analyze or specify the behavior of particular systems. Specification, the production of a precise description of an existing or intended system, has become almost synonymous with "formal methods."

14.3.8 Kinds of Notation

The formal notations that have been influential in computing fall into two main camps. The first is *finite process*—notations capturing the sequences and processes of computation in terms of a finite number of stages, phases, states, or steps. This includes many diagrammatic representations such as state-transition networks and flowcharts, and also textual notations such as formal grammars and production rules.

The second is *infinite state*—more obviously mathematical notations using either set and function notation, or a more "algebraic" representation. The most common of these is "Z," which was originally developed by the programming research group at Oxford (Spivey, 1988).

One of the enduring problems in computing has been representing and reasoning about *concurrent activity*. One thing at a time is difficult enough, but several things happening at once spell analytic breakdown! However, this is particularly important for HCI, as even a simple interface involves the human and computer working together. Collaboration and networking add to this,

In fact, one of the oldest diagrammatic notations in computing is Petri Nets, which was designed to represent concurrent and parallel activity (Petrri, 1962; see also Petri Nets World). This has a small but continuing and dedicated community. Probably more influential (possibly because they look more like programming languages) have been the more algebraic notations for concurrency, CCS (Calculus of Communicating Systems) and CSP (Communicating Sequential Processes) (Hoare, 1985; Milner, 1980). These cover very similar ground to one another and, perhaps because of this, have been the source of some of the deepest internecine strife in computing. CCS has been the basis for an ISO standard notation LOTOS (1989). Both Petri Nets and LOTOS (Language of Temporal Ordering Specifications) have been used extensively in HCI.

14.4 DETAILED DESCRIPTION

In this section, we'll look at how different aspects of mathematical, formal, and algorithmic methods can be used in HCI.

14.4.1 Two Plus Two—Using Simple Calculation

Straightforward mathematical calculations are everywhere in HCI. These range in complexity. At the simplest are simple arithmetic, for example, in the Goals, Operators, Methods, and Selection (GOMS) keystroke-level model (Card, Moran, & Newell, 1980, 1983). The models behind information-foraging theory (see Chapter 7) are more complex, using differential equations. In areas such as visualization, information retrieval, and graphics, mathematics is again central.

Even small, "back of the envelope" calculations can be surprisingly effective in helping one to understand a problem. I recall some years ago thinking through the statement (which I'm sure you've seen as well) that graphical displays have high "bandwidth." This obviously has to be interpreted in terms of visual perception, not just raw pixels per second, but I'll accept it for output. But

Compare the keyboard with the screen for rate of entry measured in bits per second.

Keyboard:
Take typing times from KLM times quoted in Dix and colleagues (1998).

```
nos targets – 64 keys
good typist – 9 keys per sec.
rate = 9 * log2(64) = 54 bps
```

Screen:
Screen width W with items size S on it. The average distance to target is half the width. To make calculations easier, assume a square screen and that the screen is completely filled with targets.

Fitts' Law—$0.1 \log_2 (D/S + 0.5)$
$D = W/2$
nos items—$(W/S)^2$

$$\text{rate} = \frac{\log_2 ((W/S)^2)}{0.1 \log_2 (W/2S + 0.5)}$$
$$\approx \frac{2 \log_2 (W/S)}{0.1 \log_2 (W/S)}$$
$$= 20 \text{ bps}$$

So, screen clicking is nearly three times slower than typing!

BOX 14.4 Back of the Envelope

what about input—do screen buttons and icons increase that? In fact, a quick Fitts' Law calculation shows that no matter what the number and size of the screen buttons, a reasonable typing speed is always faster (see Box 14.4). The difference is that, whereas the lexicon of the keyboard is fixed and has to be interpreted by the user in any context, graphical user interfaces (GUIs) can be contextual, showing appropriate actions. (If you know any information theory, this is a form of adaptive compression.) Notice that a small mathematical argument can lead to a design perspective.

Let's work through a similar example. Often the 7+/−2 rule (Miller, 1956), which is about working memory, is mistakenly overapplied. One example is for menu systems; you may well have seen suggestions that the number of menu items per screen (e.g., on a Web page) shouldn't exceed 7+/−2. On a touch screen, large targets are easier to hit, again suggesting that small numbers of larger targets are a good idea. However, the fewer menu items on a single screen, the more menu levels that are required to find particular content. Let's assume there are N items in total, and you choose to have M menu items per screen. The depth of the menu hierarchy, d, is given by:

$$d = \log N / \log M$$

If we look at a single display, the total time taken will be the time taken to physically display the screen and the time taken for the user to select the item, all times the number of levels:

$$T_{total} = (T_{display} + T_{select}) \times d$$

Using Fitts' Law for T_{select} and the formula for d, we get:

$$\begin{aligned} Ttotal &= (T_{display} + A + B \log M) \times \log N / \log M \\ &= ((T_{display} + A) / \log M + B) \times \log N \end{aligned}$$

Notice that the effect of the increased number of screens exactly balances the gains of larger targets, and the only factor that varies with the number of menu items is the per-screen costs ($T_{display}$ + A). This suggests that the more items per screen the better. Look at virtually any portal site, and you'll see that practical experience has come to the same conclusion!

In fact, there are extra factors to consider; for very small targets, Fitts' Law starts to break down, which puts lower limits on target size. Also, errors are significant, as they cause wasted screen displays, so smaller numbers of well-explained items may be better. For larger numbers of items, a further factor sets in—the time taken for the user to locate an item on the display. There is evidence that, for linear menus, the actual select time is closer to linear than logarithmic. Redoing the calculations shows that this visual search becomes the limiting factor for larger screens, leading to a maximum menu size depending on refresh time (which is still much larger than 7+/−2 for most cases!) However, good design of screen organization and subheadings can push this visual search back toward the logarithmic case (see (Larson & Czerwinski, 1998). For WAP (wireless application protocol) with small scrolling displays, the figures are again different.

Notice that being precise forces us to make assumptions explicit, and also, by focusing us on the critical factors, helps us look for potential design solutions.

14.4.2 Detailed Specification

One of the main uses of formal methods in HCI, as in other areas of computing, has been to specify the behavior of specific systems. There are three reasons for doing this:

- to analyze the system to assess potential usability measures or problems;
- to describe the system in sufficient detail so that the system that is implemented is what the designer intends;
- as a side effect of the previous item, the process of specification forces the designer to think about the system clearly and to consider issues that would otherwise be missed.

Some of the earliest work in this area used formal grammars, in particular Reisner's work on BNF (Backus-Naur Form) (1981). The strand of work arising from this is described in detail in Chapter 6.

Closely related to this is the widespread use of diagrammatic representations of dialogue structure, including the uses of STNs (as in Section 14.2). Diagrammatic formalisms are often seen as being less formal, with implications (depending on your viewpoint) of being (1) less rigorous or (2) more accessible. It is certainly the case that, for many people, diagrams are more immediately appealing, but they can be just as rigorous as more obviously "mathematical"-looking notations (see Box 14.5). The main "informality" of many graphical notations is in the labels for boxes and arcs, which are typically important for understanding the real meaning of the diagram, but not interpreted within the formalism. As we'll see in the case study in Section 14.5, this is actually a very powerful feature.

As noted earlier, human–computer dialogue typically involves multiple, potentially concurrent strands of activity. Even something as simple as a dialogue box may have many button groups, each easy to discuss individually, but all can be used in an arbitrary order by the user. The HCI group at Toulouse, France, pioneered the use of an object-based variant of Petri Nets called, ICO (Interactive Cooperative Objects) for specifying user-interface properties (Palanque & Bastide, 1995). Recall that the Petri Net formalism was designed precisely to be able to manage concurrent activity. Petri Nets have a long-standing analytic theory, which can be used to analyse the resulting representations of systems, used both in Toulouse and by others who have adopted Petri Nets. The Toulouse

The idea that graphical equals informal would have been very strange to the Greek geometers and many (not all) mathematicians for whom diagrams are critical for understanding. When Newton wrote his treatise on gravity and motion, *Principia Mathematica,* he deliberately used geometric explanation rather than more textual notations, because he thought the mathematical explanation would make things too easy. Only really clever people would be able to understand the diagrams!

BOX 14.5 Newton's *Principia Mathematica*

group have worked for some years using their formalism in the design of air-traffic-control interfaces and have recently created design and analysis tools for the approach (Navarre et al., 2001).

Another successful approach to concurrency has been the use of LOTOS at the CNUCE Institute in Pisa, Italy. It is interesting to note, however, that this work has found greater external interest since the emphasis shifted from full use of the LOTOS notation to a diagrammatic representation of task hierarchies, called ConcurTaskTrees (CTT), where groups of subtasks are linked using concurrency operators from LOTOS (making more precise the plans in standard HTAs (Hierarchical Task Analysis) [Shepherd, 1995]). The original intention of this was as a bridge toward full use of LOTOS, but it has gradually stolen the limelight. Again, a critical feature has been the introduction of a support tool CTTE (ConcurTaskTrees Editor) (Paternó, 2000; Paternó, Mori, & Galimbert; 2001; Paternó & "Santoro, 2001).

These are all notations and formalisms where the principal focus is on the flow of dialogue. Others have used state-oriented notations such as Z to specify systems. One of the earliest examples of this is Sufrin's (1982) specification of a "display editor" (in contrast to command line) back in 1982. Algebraic notations have also been used (Torres et al., 1996). I find that these full specifications really force one to think hard about the system that is being designed.

A simple example I use with students and in tutorials is a four-function calculator. What is in the state? Clearly there is a number that is currently displayed, but many students get stuck there. The parts that have no immediate display are harder to think about as a designer. However, this hidden state is also more confusing for the user when it behaves unexpectedly, so it is more important that the designer gets it right.

user types: 1 + 2 7 = − 3

start after 1 + 2

action	display	pend_op	total
	2	+	1
digit(7)	27	+	1
equals	28	none	28
op(−)	28	−	28
digit(3)	**283 !!!**	−	28

BOX 14.6 Calculator Scenario

Some of the extra detail becomes apparent when you think about particular user actions—when the = key is pressed, the calculator must know what operation to do (+,−,*,÷) and what the previous value was, so the state must record a "pending" operation and a running total.

The other method I find useful is to play through a scenario annotated with the values of various state variables. Doing this to the calculator (see Box 14.6) shows that an additional flag is needed to distinguish the case when you just typed "2" from when you just typed "1+1." In both cases the display says "2," but in the former case typing an additional "3" would give you "23," whereas in the latter case it would be "3."

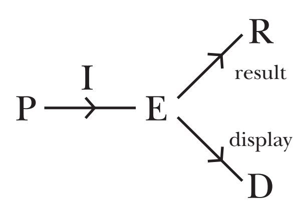
Of course, formal notations are not very useful unless they give some benefit. In the hands of an expert analyst, the process of producing a detailed specification will reveal new insights into the usability of the system. However, formal specification also allows the opportunity for more automated analysis. Automatic proof tools have been used, but they are still difficult to use. Because of this, there has been a growing interest in model checking tools that enable (still quite simple) properties to be checked automatically (Abowd, Wang, & Monk, 1995; Paternó & Santoro, 2001). One advantage of these is that, if a property fails, then the normal output of the checking tool is a trace of user actions that would cause the violation—a crude scenario that can be fed back to those not expert in formal techniques.

Dialogue specifications were used extensively in the early user-interface management systems (UIMS) (Pfaff & ten Hagen, 1985). For a period, these were less extensively used, partly because the ready availability of event or object-based GUI builders lead to a more "try it and see" model of UI construction. However, there has been a growing research agenda in computer-aided design and construction of user interfaces (for example, the CADUI conference series).

A significant area within this has been "model-based" user-interface-development environments including UIDE (Foley & Sukaviriya, 1995), MASTERMIND (Szekely, et al., 1996), and TADEUS (Elwert & Schlungbaum, 1995). These have some sort of model of the user's task, the underlying application semantics, and the device capabilities. The designer creates the models using a notation or design environment, and then the system automatically generates the actual user interface. This is particularly powerful when targeting multiple devices—for example, PDA (Personal Digital Assistant), WAP, Web, as well as standard desktop interfaces.

14.4.3 Modelling for Generic Issues

The area of formal methods with which I am most associated personally, and which perhaps is the beginning of formal methods in HCI as an identifiable area, is the strand of modeling started in York, England, in the mid-1980s, typified by the PIE model (Dix & Runciman, 1985).



The PIE model takes a general view of an interactive application as a transformer of the history of user actions (for historical reasons labeled P) into a current display and possibly also some kind of additional result (printout, saved document). This model and variants of it were used to specify and analyze a range of general user interface issues, mainly to do with the users' ability to predict future behavior based on the available display and their ability to reach desired goals in the system (Dix, 1991).

This use of formal methods is particularly powerful, because one is able to make inferences about properties of a whole class of systems. The cost of working with very precise models is defrayed over many systems, which are instances of the general class.

Other examples of this kind of activity include work on understanding case-based systems as used in help desks (Dearden & Harrison, 1995), and the modeling of authoring environments for educational media (Kotze, 1997). Abstract formal modeling has even been used to formalize some of the cognitive dimensions properties found in Chapter 5 (Roast & Siddiqi, 1996).

A topic that has been studied intermittently from the very first PIE paper [[DR85]] onwards has been **Undo.** This is partly because it is a generic issue and so a good one to deal with using such models, and partly because the nature of **Undo** makes it a good candidate for formalization. Some systems try to make **Undo** reverse the effect of any previous command, including itself. Quite early on, we were able to show that (with the exception of two-state systems like light switches!) this was impossible and that **Undo** has to be regarded as a different kind of meta-action. There has been a substantial amount of work since then, including group **Undo** (see Section 14.4.4). However, the most complete treatment of **Undo** has been in Mancini's thesis (1997), which captures the multilevel nature of **Undo.** The same model, 'the cube', based on a form of the PIE model, was also used to model "back" and "history" mechanisms in hypertext and the Web, showing how formalism can expose and capitalize on similarities between disparate phenomena (Dix & Mancini, 1997; Dix, Mancini & Levialdi, 1997).

14.4.4 Computer-Supported Cooperative Work and Groupware

Formal modeling and specification work has not been limited to single user interaction. As well as specifications of particular systems, there are a number of areas that have had more extensive analysis. One example is workflow packages, which are based on various kinds of models of collaborative processes. Here the formal representation is largely in order to build a system, which is reminiscent of the use of formal dialogue notations in early UIMS.

An area that has required deeper formal treatment is distributed editing and group **Undo.** The problems with the latter are mainly due to "race" conditions when two people edit parts of the document virtually simultaneously, and messages between their computers pass one another on the network (Dix, 1995a; Ellis & Gibbs, 1989). Group **Undo** problems are perhaps more fundamental as there are similar implementation issues, and it is also unclear what is the right thing to happen anyway! (See Abbowd & Dix, 1992).

Although there have been a number of papers dealing with specifications of particular computer-supported cooperative work (CSCW) issues (e.g. Palanque & Bastide, 1996), there are few notations aimed specifically at this area. CTT is an exception, as it has been extended to explicitly deal with several people cooperating on a task (Paternó, 2000). My own LADA (A Logic For The Analysis of Distributed Action) notation is another example (Dix, 1995b). As well as using formal methods to talk about the groupware systems themselves, there has been some work in using formalisms to understand the human side of collaboration. Winograd and Flores' (1986) use of speech-act theory embodied in the Coordinator tool is one example. Belief logics and related formalisms (Devlin, 1994; Ellis & Wainer, 1994) have also been used to formalize the shared and disparate

Alison and Brian are working together on a report using a collaborative editing application. Alison pastes a paragraph into the report, realizes she has not put it where she wants, and is about to select **Undo** when the phone rings. While she is on the phone, Brian, in his office, notices that the paragraph looks a little strange and so edits it and the surrounding text to make it flow. Alison finishes her phone call and, without looking at her screen, continues where she left off and presses **Undo.** What happens? Does this undo Brian's last action? Does it remove the paragraph she pasted, in which case what happens to the edits that Brian has done to it?

BOX 14.7 Group Undo

beliefs and knowledge of collaborating people. This could be seen as a formal parallel of the sort of shared understanding described in Chapter 10. Another example of this type of work has been the use of artificial life and game theory to understand collaboration (Marsh, 1994).

14.4.5 Time and Continuous Interaction

In applied mathematics, for example simple mechanics, one typically deals with continuous phenomena such as the location and velocity of a ball:

$$\frac{dv}{dt} = -g \qquad \frac{dx}{dt} = v$$

In contrast, computing tends to deal with discrete values at clocked time points. Instead of rates of change, we look at state changes at particular events or on clock ticks. Certain types of system include discrete computer controllers interacting with continuous real world phenomena, for example, a fly-by-wire aircraft or an industrial controller. A branch of formal specification has grown up to deal with such phenomena, called *hybrid systems* (Grossman, et al., 1993), but it is very much a minority area compared with the discrete formal-methods community. The hybrid systems area also draws a sharp distinction between discrete computer components and continuous physical ones.

S
R
L

Even in desktop interfaces, there are phenomena that are continuous, such as the movement of the mouse, and others that we perceive as continuous, such as the location of an icon we are dragging. For many years, I have argued that these status phenomena are important and should be dealt with on a par with event phenomena (Dix 1991a, 1991b, 1196, 1998). More recently, virtual reality, ubiquitous computing, and similar areas have further emphasized the importance of these phenomena, and various strands of modeling work have ensued—including a European Union project, TACIT, focused on continuous phenomena, which used notations from the hybrid systems area (Doherty, Massink, & Faconti, 2001) and other work based on systems theory (Wûthrich, 1999).

A related area that has always been a hot topic personally is time. Delays and timing have long been seen as important in HCI (Schneiderman, 1984). However, there was a period from roughly the mid-1980s to mid-1990s when it was assumed that personal computers were getting faster and faster and so problems due to time would go away. This led to implicit design assumptions I called the "myth of the infinitely fast machine" (Dix, 1987). It was an area I have returned to several times in formal and semiformal material (Dix, Ramduny, & Wilkinson, 1998) until, largely due to delays on the Web, it became widely clear that computers wouldn't just get "fast enough." Recent formal work in the area has included attempts to deal with multiple granularity in time (Kutar, Britton, & Nehaniv, 2001).

14.4.6 Paradigms and Inspiration

There has been a rich interplay between cognitive and computational models since the earliest days of computing. Cognitive models have influenced artificial intelligence (AI) research and neural networks, and AI models and computer-like architectures have been used to model human cognition. There is an even longer Pygmalion tradition of constructing humanoids in clockwork, steam, and flesh enlivened with electricity—looking to the latest technology in the hope of capturing the essence of the animus that makes us human. The human as computer model is perhaps simply the latest in this line, but it has been informative and successful despite that.

Computational analogies can similarly be useful as a way of understanding various interface phenomena.

If we assume users have perfect knowledge of the system and its behavior, they should be able to work out an optimal route through the system to their goal. This can be seen as an AI-planning model of human interaction and is the implicit model behind a variety of optimal-path interaction metrics (Thimbleby, Cairns, & Jones, 2001).

However, much of interaction is more contingent. We sort of know what we want, and when we see it we may recognize it or at least see that one state of the system is closer than another. Our interaction with the system is more of a goal-seeking behavior, seeking a partly understood goal within a partly understood system. For such systems, search and optimization algorithms may be better models than planning.

Search and optimization is a huge area within computing, and it is complex partly because many of the problems are intractable (in the sense of exponential time)—you cannot consider every possible solution. One class of methods is called *hill climbing*. You take a partial solution and look at all closely neighboring solutions. If all of them are less good, then your current solution is (locally) the best. If any are better, you take the best of the solutions you considered as your new current solution. This is like walking in the hills and always taking the steepest upward path. Eventually you come to a hill top. The problem with steepest-ascent hill climbing is that you are only guaranteed to get to some hill top, not the biggest. If you start in Cambridge, in England, you are likely to end up at the top of the Gog-Magog hills (approximately 100 metres, or 300 feet); if you start at Kathmandu, the peak you end up at will be much higher! Locations such as the Gog-Magog hills, which are higher than anything close, but not highest of all, are called *local maxima*.

One solution to this, which has been very powerful in AI game-playing algorithms, is to find a *heuristic* (Finlay & Dix, 1996). This tells you not how good a partial solution is, but how good further solutions in a particular direction are likely to be. For example, in chess you can score a board position based on the value of the pieces. With a suitable heuristic, one can perform hill climbing (or variants), but based on the heuristic rather than on the actual value. In real hill climbing, instead of looking at the heights just one step away, one scans the horizon to see which direction looks more mountainous.

In a user interface, this reminds us how important it is for users to be able to predict what will happen if they perform actions. Direct manipulation and graphical interfaces are particularly bad at this because vision is a here-and-now sense (see my discussion of visual versus "nasal" interfaces [Dix, 1996]). Surprisingly few visualisation systems give you this sort of "over the horizon" view to enable a heuristic search. Two exceptions are HiBrowse (Ellis, Finlay, & Pollitt, 1994), which allows you selectively drill down hierarchical attributes in a database, telling you how many items would be selected if you choose particular values; and Influence Explorer (Tweedie, et al., 1995), a dynamic slider-based visualization that has histograms of selected items and also (differently colored) items that just miss one or two selection criteria. At a broader level, information scent (Chapter 7) is a form of heuristic.

Another good solution to avoid local maxima in the computer domain is adding randomness, either just starting at random locations and doing a simple hill climb, or being a bit random in one's exploration, not *always* taking the best path (e.g., simulated annealing). Thimbleby and colleagues (2001) model interfaces as finite graphs and use various graph theoretic properties to analyze potential usability. Their methods include constructing Markov models, which measure how likely users are to make particular state transitions (or perform particular actions depending on the current state). In various interfaces, they find that random strategies perform better than typical user behavior, because users have mistaken models and hence planning strategies fail! This is also one of the reasons why children perform better with computer systems and household appliances than adults—they are less afraid to experiment randomly. (See my randomness Web pages for more about algorithms that exploit stochastic properties [Dix, 2002b]).

Search is just one example where we can draw analogies between algorithms and interfaces to give us inspiration for design (and for algorithms). However, search is special, not just because goal-directed exploration is a general interface issue, but also because the process of design itself is a form of goal-directed search.

Consider the normal pattern of prototyping-oriented design:

1. think of an initial design
2. evaluate it
3. consider alternatives to fix bugs or improve the design
4. select the best
5. evaluate that
6. . . . iterate . . .

In fact, this is precisely hill-climbing behavior and has exactly the same problems of local maxima—designs that cannot be incrementally improved but are not necessarily that good.

A really good designer will have some idea of the final solution, based on experience and insight; so the initial start point is more likely to be a "Kathmandu" prototype than a "Cambridge" one.

Given not all designers are so inspired, traditional graphical design practice usually includes several diverse start-points before commencing more incremental improvements—a form of hill climbing with random start-points.

However, it is analytic and theoretical approaches that can give us the ability to have heuristics, assessing more accurately the potential directions. This is particularly important in novel areas where we have little experience to guide our intuition.

So, if you are a brilliant designer in a well-trodden area—do what you like. Otherwise, a prototyping-based UI development process is essential as we don't understand users or interfaces well enough to produce the right system first time, but prototyping is fundamentally flawed unless it is guided by an analytic and theoretical framework.

14.4.7 Socio-Organizational Church-Turing Hypothesis

I'll conclude this section by discussing another area where computation can act as a paradigm or inspiration.

An organization has many aspects: social, political, financial. However, among these diverse roles, many organizations perform an information-processing role: transforming orders into delivery instructions and invoices, student exams into degree certificates. Recall that the Church–Turing thesis postulates that all computation is fundamentally equivalent. This only covers what is possible to achieve, not how it happens, which of course is very different!

However, practical experience shows that all practical computational devices tend to exhibit similar structures and phenomena. This is why models of physical computation have proved so effective in cognitive modeling and vice versa. If we apply this to an organization, we get the *socio-organizational Church–Turing hypothesis,* a term coined by Wilkinson, Ramduny-Ellis and myself at a workshop in York some years ago (Dix, Wilkinson, & Ramduny, 1998; Dix, 2002c). If we look at the information-processing aspects of an organization, we would expect to see similar structures and phenomena to those in mechanical computation devices

For example, computers have different kinds of memory—long-term memory (hard disks) and short-term memory. (RAM and processor registers). In an organization, the former is typically explicit in terms of filing cabinets of paper or computer systems, and the latter is often held in people's heads or scribbled in notes on jotters.

A special part of the memory of a computer system is the program counter, which says what part of a program the computer is currently executing. The organizational analogy is the state of various formal and informal processes. If we look for these *placeholders* for the state of organizational processes, we typically find them either in people's memory or to-do lists, or in the physical

	computational parallels		
	computer	**cognitive model**	**organization**
process	program	procedural memory	processes
data	data	long-term memory	files
placeholder	program counter	short-term memory/activation	location of artifacts
initiative	interrupts, event-driven	stimuli	triggers

TABLE 14.2 Structural parallels of the socio-organizational Church-Turing hypothesis.

placement of papers and otherartifacts. That is, the office ecology is part of the organization's working memory; cleaners who tidy up are brainwashing the organization!

Table 14.2 summarizes the structural parallels between computers, cognitive models, and organizations.

14.5 CASE STUDY—DIALOGUE SPECIFICATION FOR TRANSACTION PROCESSING

As my case study, I'm going to use not a recent example, but one from more than 15 years ago—in fact, from before I became a computing academic, and before I'd even heard the term HCI! At the time, I was working for Cumbria County Council, in England, working on transaction-processing programs in COBOL, the sort of thing used for stock control or point-of-sale terminals in large organizations.

Why such an old example, rather than a more sexy and up-to-date one? Well, first, because this sort of system is still very common. In addition, the issues in these large centralized transaction-processing systems are similar to those of Web-based interfaces, especially e-commerce systems. Third, it is a resounding success story, which is not too common in this area, and a 1000% performance improvement is worth shouting about. Finally, and most significantly, because it was a success it gives us a chance to analyze why it was so successful and what this tells us about using formalism today.

The other thing I ought to note is that, although this was a successful application of formal methods in interface design, I didn't understand *why* at the time. It is only comparatively recently that I've come to understand the rich interplay of factors that made it work and so perhaps be able to produce guidelines to help reproduce that success.

14.5.1 Background—Transaction Processing

Transaction-processing systems such as IBM CICS have been the workhorses of large scale information systems, stock management, and order processing since the late 1970s. They are designed to be able to accept many thousands of active users.

Architecturally, these systems are based around a central server (or cluster) connected to corporate databases and running the transaction-processing engine (see Figure 14.4). In the system I worked with, this was an ICL mainframe; in Web-based applications, however, it will simply be a Web server or enterprise server. The user interacts with a form-based front end. In the systems I dealt with in the mid-1980s, the front end was semi-intelligent terminals capable of tabbing between fields. Subsequently, in many areas these were replaced by personal computers running "thin client" software and now are often Web-based forms. The centralization of data and transaction processing ensures integrity of the corporate data, but the fact that users interact primarily with local terminals/PCs/browsers means that the central server does not have to manage the full load of the users' interactions.

When the user interacts, the succession of events is as follows:

1. user fills in form on terminal,
2. terminal may perform some initial validation (e.g., numbers vs. letters, range checks, date format, or, on thin PC client or Javascript on Web forms, more complex validation);
3. user checks and then submits form (presses special key or screen button);
4. terminal/PC/browser sends form data as a message to the transaction-processing engine (e.g., CICS or Web server) on the central server;
5. transaction-processing engine selects appropriate application module for message (based on last screen/Web page or information in message);
6. application module interprets message (form data), does further checks, performs any database updates, gets any required data from the database, and generates a new screen/Web page as result
7. transaction processing engine passes this back to the terminal;
8. terminal presents the screen/Web page to the user.

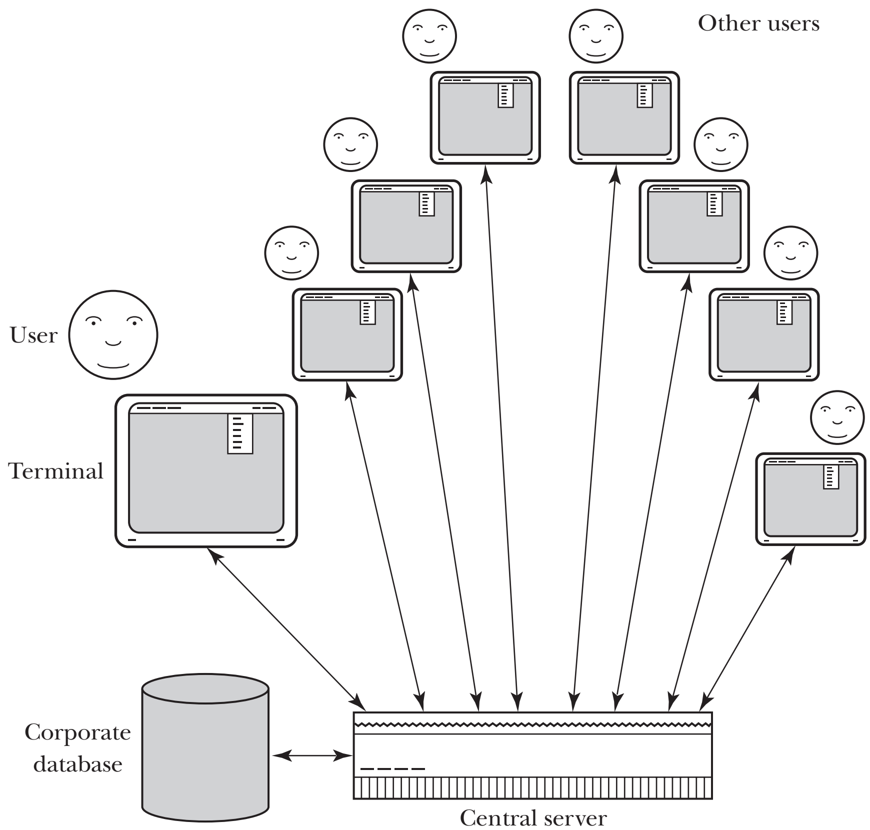


FIGURE 14.4 Physical architecture of a transaction-processing system.

All these stages except the sixth one are managed by the transaction-processing infrastructure. This sounds as if the job in designing this part should be straightforward, as if most of the complexity of dealing with detailed user interactions has been dealt with. But it is not quite as simple as all that.

14.5.2 The Problem . . .

In a graphical user interface (GUI) or any single user interface, the succession of events in the program is straightforward:

1. user event 1 arrives (e.g., mouse press)
2. deal with event and update display
3. user event 2 arrives (e.g., mouse release)
4. deal with event and update display
5. user event 3 arrives (e.g., key click)
6. deal with event and update display

As we know, this can cause enough problems!

In a transaction-processing system, with one user, the application module may receive messages (with form data) in a similar fashion. However, the whole point of such systems is that they have many users. The module may therefore receive messages from different users interleaved:

1. message 1 for user A received
2. deal with message and generate new screen/Web page for user A
3. message 1 for user B received
4. deal with message and generate new screen/Web page for user B
5. message 2 for user B received
6. deal with message and generate new screen/Web page for user B
7. message 2 for user A received
8. deal with message and generate new screen/Web page for user A

The transaction-processing engine deals with passing the new screens back to the right user, but the application module has to do the right things with the form data in the messages. In the case of simple transactions, this may not be a problem; for example, if the application simply allows the user to enter an ISBN (International Standard Book Number) for a book and then returns data about that book, the system can simply deal with each message in isolation. However, a more complex dialogue will require some form of state to be preserved between transactions. For example, a request to delete a book may involve an initial screen where the user fills in the ISBN, followed by a confirmation screen showing the details of the book to be deleted. Only then, if confirmed, will the system actually do the deletion and generate a "book has been deleted" screen. Even a search request that delivers several pages of results needs to keep track of which result page is required and the original search criteria.

Getting back to my workplace in Cumbria in the mid-1980s, the transaction systems in place at that stage dealt only with the simple stateless record-display transactions or multipage search transactions . . . and even the latter had problems. When several users tried to search the same database using the system, they were likely to get their results mixed up with one another!

I was charged with producing the first update system. Whereas occasionally getting someone else's search results was just annoying, deleting the wrong record would be disastrous.

14.5.3 All About State

So what was wrong with the existing systems, and how could I avoid similar but more serious problems? In essence, it is all about state.

In most computer programs, you don't need to worry too much about state. You put data in a variable at one stage; if you require the data at a later point it is still there in the variable. However, in the case of transaction-processing modules, the module may be reinitialized between each transaction (as is the case with certain types of Web CGI (Common Gateway Interface) script), so values put in a variable during one transaction won't be there at all for the next transaction. Even worse, if the same module is dealing with several users and not reinitialized, then values left behind from a transaction for one user may still be "lying around" when the next user's transaction is processed. This is precisely what was happening in the search-result listings. Some of the state of the transaction (part of the search criteria) was being left in a variable. When the system was tested (with one user!), there was no problem, but when several users used the system their search criteria got mixed up. Although it was possible to explicitly save and restore data associated with a particular terminal/user, the programmers had failed to understand what needed to be stored. Instead, the existing programs coped by putting state information into fields on the form that were then sent back to the next stage of the transaction. With Web-based interfaces, similar problems occur with session state.

However, there is also a second, more subtle part of the state: the current location in the human–computer dialogue.

In traditional computer algorithmics, the location in the program is implicit. It is only when one starts to deal with event-driven systems, such as GUIs, network applications, and transaction processing, that one has to explicitly deal with this. And of course traditional computer-science training does little to help. Not only are the principal algorithms and teaching languages sequential, but also the historical development of the subject means that sequential structures such as loops, recursion, and so on, are regarded as critical and are in lists of essential competency, whereas event-driven issues are typically missing. Even worse, event-based languages such as Visual Basic and other GUI development languages have been regarded as "dirty" and not worthy of serious attention. Possibly this is changing now that Java is becoming a major teaching language, but still the event-driven features are low on the computer-science agenda!

So, computer programmers in the mid-1980s—as well as those of today—were ill prepared both conceptually and in terms of practical skills to deal explicitly with state, especially flow of control.

This was evident in the buggy transaction modules I was dealing with. The flow of the program code of each module looked rather like a twiggy tree, with numerous branches and tests that were effectively trying to work out where in the flow of the user interaction the transaction was situated.

```
if confirm_field is empty // can't be confirm screen
               // or user didn't fill in the Y/N box
then if record_id is empty // must be initial entry
     then prepare 'which record to delete' screen
     else if valid record_id
          then read record and prepare confirm screen
          else prepare error screen
else if confirm_field = "Y'
     then if record_id is empty // help malformed
          then prepare error screen
          else if valid record_id
               else do deletion
               then prepare error screen
     else if confirm_field = "N'
          then prepare 'return to main menu' screen
          else prepare 'must answer Y/N' screen
```

No wonder there were bugs!

Of course, if one looks at many GUIs or Web applications, the code looks just the same—Try using the **Back** key or bookmarking an intermediate page in most multistage Web forms, and you'll probably find just how fragile the code is.

14.5.4 The Solution

A flow chart of the program looked hideous and was very uninformative because the structure of the program was not related to the structure of the user interaction. Instead of focusing on the code, I instead focused on the user interaction

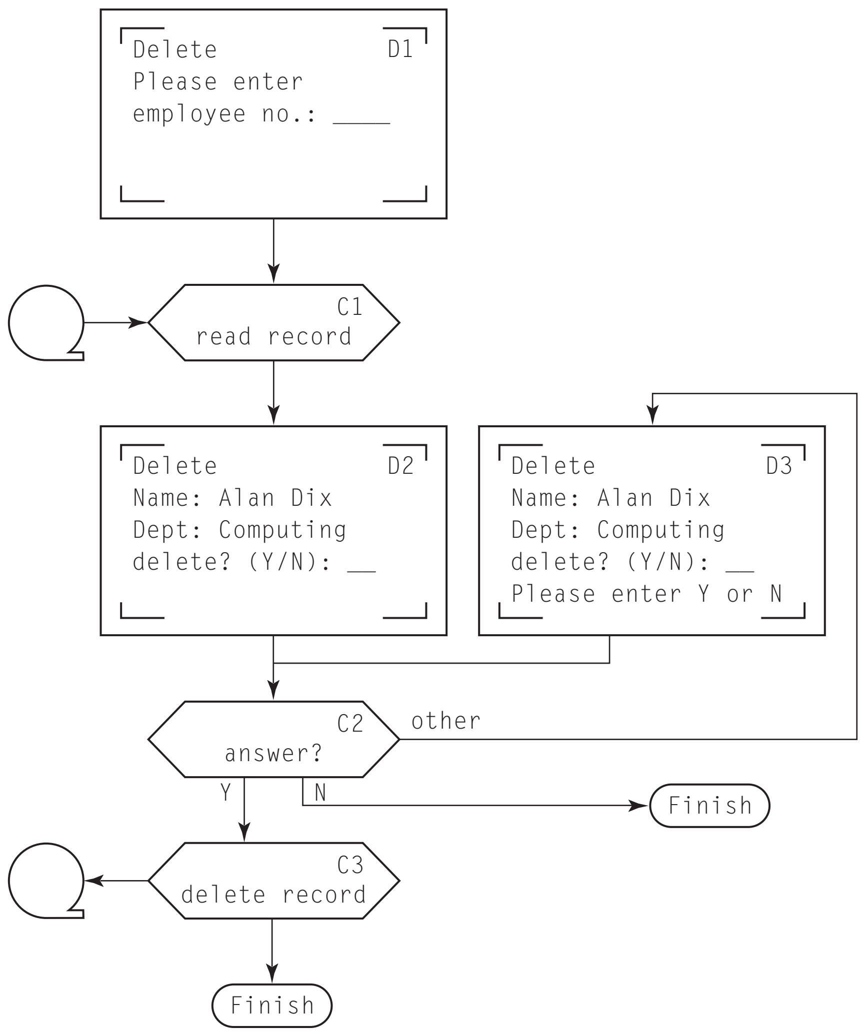


FIGURE 14.5 Flow chart of user interaction.

and produced flowcharts of the human-computer dialogue. Figure 14.5 shows a typical flowchart. Each rectangle represents a possible screen, and a miniature of the screen is drawn. The hexagonal boxes represent system actions, and the "tape" symbols represent database transactions. Note that this is not a flowchart of the program, but of the human-computer dialogue, rather like the STNs in

Section 14.2. Note also that the purpose is to clarify the nature of the dialogue, so the system side is only labeled in quite general terms (e.g., "read record"). These labels are sufficient to clearly say what should happen and do not say how the code to do this works in detail. This is because the difficult thing is getting the overall flow right.

Notice also that each major system block and each screen is labeled: D1, D2, D3 for the screens; C1, C2, C3 for the system code blocks. These are used to link the flowchart to boilerplate code. For each screen there is a corresponding block of code, which generates the screen and, very important, stores the label of the next system block against the terminal/user. For example, screen D3 will save the label C2. The first thing the module does when asked to deal with a message is to retrieve the label associated with the user. If this is blank, it is the start of the dialogue (it will generate screen D1 in this case); otherwise the module simply executes the code associated with the relevant system block.

This all seems very mundane, but the important thing is that it worked. Systems that were taking months to develop could be completed in days, and the turnaround time for upgrades and maintenance was hours. That is, systems were being produced at least 10 times faster than previously and, furthermore, with fewer bugs!

14.5.5 Why It Worked . . .

So why is it that such a simple formal method worked so well, and can we use this to assess or improve other formalisms or develop new ones?

Let's look at some of the features that made this method function well:

***useful**—addresses a real problem!*

The notation focused on the overall user-interface dialogue structure that was causing difficulties in the existing systems. So often formalisms are proposed because they have some nice intrinsic properties, or are good for something else, but they do not solve a real need.

***appropriate**—no more detailed than needed*

For example, there was no problem in producing the detailed code to access databases and so forth, so the formalism deals with this at a crude level of "read record," "delete record," and so on. Many formalisms force you to fill in lots of detail, which makes it hard to see the things you really need them for as well as increasing the cost of using them.

communication*—mini-pictures and clear flow easy to talk through with client*

Formal methods are often claimed to be a means to improve communication within a design team, because of their precision. However, when precision is achieved at the cost of comprehensibility, there is no real communication.

complementary*—different paradigm than implementation*

It is quite common to use specification methods that reflect closely the final structure of the system, such as object-oriented specification for object-oriented systems. Here, however, the specification represents the structure of the dialogue, which is completely different from the structure of the code. This is deliberate: The notation allows one to see the system from a different perspective, in this case one more suitable for producing and assessing the interface design. The relationship between the structure of the notation and the structure of the code is managed via simple rules, which is what formalisms are good at!

fast pay back*—quicker to produce application (at least ten times faster)*

I have been involved in projects where substantial systems have been fully specified and then implemented, and I have seen the improvements in terms of quality and *long-term* time savings. However, I still rarely use these methods in practice even though I know they will save time. Why? Because I, like most people, like instant payback. Spending lots of time up front for savings later is very laudable, but when it comes to doing things I like to see results. Not only that, but clients are often happier to see a buggy partial something than to be told that, yes, in a few months it will all come together. The dialogue flowcharts didn't just produce long-term savings, but they also reduced the lead time to see the first running system.

responsive*—rapid turnaround of changes*

The feeling of control and comprehension made it easier to safely make changes. In some formal methods, the transformation process between specification and code is so complex that change is very costly (see Dix & Harrison [1989] for a discussion of this). The assumption underlying this, as in much of software engineering, is that well-specified systems will not need to be changed often. Of course, with user interfaces, however well specified, it is only when they are used that we really come to fully understand the requirements.

reliability—*clear boilerplate code is less prone to errors*

Although the transformation process from diagram to code was not automated, applying and modifying boilerplate code templates was a fairly automatic hand process. This heavy reuse of standard code fragments greatly increases the reliability of code.

quality—*easy to establish test cycle*

The clear labeling of diagrams and code made it easy to be able to track whether all paths had been tested. However, note that these are not just paths through the program (which effectively restarted at each transaction), but each path through the human–computer dialogue.

maintenance—*easy to relate bug/enhancement reports to specification and code*

The screens presented to the user included the labels, making it easy to track bug reports or requests for changes both in the code and specification.

In short, the formalism was used to fulfill a purpose; it was, above all, neither precious nor purist!!

14.6 CURRENT STATUS

To see where formal methods are going in HCI, we'll start by looking back at the progress of formal methods first in computing in general and then in HCI. We'll then move on to examine the trends and potential ways to go forward in the area.

14.6.1 Retrospective—Formal Methods in Computing

There was a period, in the late 1980s and the early 1990s, when every major research project in computing *had* to have a formal methods component in order to be respectable. There were known problems in industrial take-up, but this was widely explained as due to lack of suitably trained personnel. As a new breed of formally adept graduates entered the market, it was believed, formal methods would find its true place as central to computer systems design.

Even by the end of this period, the excuses were wearing thin; certainly by the mid-1990s the popular spirit had moved against formalism: "doesn't scale," "too difficult," "requires too much training." . . . Although there is still a strong formal methods community, the emphasis is more on particular domains where safety, cost, or complexity make the effort of formal methods worthwhile. Within these areas, there have been a continual stream of industrial success stories, and some consulting firms make formal methods expertise one of their market differentiators (see Clarke and Wing's 1996 survey article.

The critique is all sensible, and "problems" in the formal methods community were largely due to overhype in the early days. In other fields, for example structural engineering, one would not expect to apply the same level of analysis to a garden gate as to the Thames flood barrier in London. Even within a single structure, the analysis of a ship's hull would be different from an internal cabin door, and a porthole beneath the waterline may be different from one above deck level.

However, there is another side to this. In so many disciplines that have been fashionable and then fallen into disrepute, one hears, usually from within the discipline itself, that "it has never been used in practice" and has failed somehow to meet (overblown) expectations. However, again and again one finds that this is because the discipline constantly redefines itself to exclude those parts that have become successful and "spun off." For example, artificial intelligence experienced the same hype and decline, but spin-off areas such as natural language processing, computer vision, and aspects of expert systems are all widely used (and used in HCI!).

Looking with less exclusive eyes, we can see a huge growth in the use of structured methods and diagrammatic specifications, especially in the use of the Unified Modeling Language (UML). Many of the notations used have their roots in what once would have been called "formal," but now is perhaps simply normal. Note too that it is precisely because systems are large and complex that more formal notations are used. It is easy for programmers to get the details right, but they need support in understanding interactions and architecture. Of course, if formal methods are defined to exclude all this, then, true, adoption is low!

14.6.2 Retrospective—Formal Methods in HCI

Within HCI, formal methods have always had a rougher ride as the rigid and cold notations stand in contrast to the richness of human interaction. The area gained somewhat due to the early general formal-methods hype, and certainly that was one of the enablers for the early York work. However, rather than growing dramatically and then suffering reverses (as was the case within computing), the formal methods subcommunity within HCI has gradually grown and is now able to support not only an annual conference largely of its own (DSVIS), but also a number of other conferences where it is a major facet (CADUI, EHCI, TAMODIA).

Although individuals have very different agendas and are wedded to their own methods, the relatively small formal methods (FM) in HCI community has tended to knit together in the face of the "opposition" from the wider HCI community and so has managed to avoid some of the internal schisms that have bedeviled FM in computing. This has meant that the collections, journal special issues, and conferences in the area have included a wide range of formalisms including logics, set theoretic specifications, diagrammatic notations, and even cognitive task models.

Formal methods has always been criticized for dealing only with toy problems, and to a large extent this would also be a valid criticism of a large proportion of the FM-in-HCI literature. A significant number of papers describe new specification techniques, developments of old ones, or applications of existing methods, and then apply them to simple case studies. This is perhaps reasonable given the need to start simple in order to address problems and issues one at a time. And there are signs of maturity, as some of the longer-established groups and methods (e.g., Toulouse and Pisa) have created support tools and are applying their techniques to problems in large commercial organizations.

We are perhaps still looking for headline success stories to sell formal methods into the mainstream HCI community in the way that the NYNEX study did for GOMS (Gray, John, & Atwood, 1992), (see also Chapter 4, section 4.5), but it seems as if this is now on the horizon.

The culture clash between postmodern HCI and formalism has always been stronger in the United States than elsewhere, and formal papers in computer-human interaction are rare. However, there has been quite a strong line of papers in CSCW conferences and HCI journals using formal arguments and analyses to deal with issues of distribution and concurrency—for example, in describing the use of "operation transforms" and **Undo**/**Redo** frameworks for managing race conditions and conflicts in group editors.[3] The various "spatial" models in CSCW are also reliant to varying degrees on formal definitions (Benford, et al., 1994; Dix, et al., 2000; Rodden, 1996; Sandor, Bogdan, & Bowers, 1997). These are

[3] For a short bibliography on operation transforms and group undo, see www.hcibook.com/alan/topics/undo.

both cases where the underlying area is complex and where simpler arguments and explanations fail.

The last of these is an example of creating generic models/specifications to address a broad issue rather than a specific system. I have always found these the most exciting papers, giving the biggest bang for buck, because the understanding gained can be applied to many different systems. However, these publications are relatively rare. Producing such work requires a comfort in using the formalisms and an ability to look at an area in broad terms. It may be that these skills are usually found in different personality types, hence making it difficult for an individual to succeed. However, it may equally well be due to education—it is only in the more "techie" courses that more formal methods are taught (and not always there). The FM-in-HCI community is therefore full of those who (and I caricature) either started off in very technical areas and are gradually learning to understand people or started off in more people-oriented areas and are gradually learning that hand waving is not enough. This of course takes quite a time to mature, but, when it does, it means that the FM-in-HCI community contains the most well-rounded people in IT!

14.6.3 Prospective

So where does this take us?

The areas that have greatest take-up are where the formalisms are simplest (as in my case study) and where there is tool support. Particularly powerful are cases where the formalism can be used to generate interfaces, either early prototypes or as part of the final running system (cf. appropriateness and fast feedback in Section 14.5). This is set to continue; for example CNUCE's CTTE is being used in a growing number of sites.

However, it is also clear that there are application areas that are intrinsically complex, such as the work on distributed group editing and group **Undo.** This area is hitting new complexity limits as the various researchers widen the types of operations they deal with; it seems that some simplifying framework is needed.

Looking more widely, the trend of having many papers about different methods is likely to continue. Each research group has its own focus, and there is a good feeling about working on "your" notation. Rather than fighting human nature, it would be good to see the development of an integrating framework to make the relationships between different notations and methods more clear. Perhaps more important, such a framework is essential to allow interoperation between the growing numbers of formally based interface design and analysis toolkits.

Note that this is still a wish, because there is no real sign of such an integrating framework at present; the closest may well still be the original York modeling work. Perhaps the closest is the work on the syndetic modeling framework (Barnard, et al., 2000), which proposes that systems are regarded as lower-level interactors each described by appropriate microtheories and then bound together into higher-level interactors by macrotheories. However, this is more a framework for formulating a broad theory, not the theory itself.

In fact, prompted by the writing of the early drafts of this chapter, I started the process of seeking a unifying framework between HCI formalisms using traces of activity as a common point between widely different notations (Dix, 2002d).

The existing safety-critical application areas, such as flight control, are likely to remain central, with perhaps other high cost-of-failure applications such as e-commerce joining them.

The Web, WAP, and other aspects of global networking are creating new problem areas that are difficult to understand:

- every system is becoming a distributed system, with portions on a user's machine, running as applets, at multiple interacting servers (consider .Net)
- multiple users, machines, and applications engender emergent properties
- state, which we have seen is difficult, is broken up over servers, URL parameters, and multiple frames and windows in a user's browser
- parts of the system are unreliable or insecure
- there are generic components (browsers, Web servers) as well as specific applications
- the **Back** button and bookmarks mean that applications keep "restarting" in the middle!

These all suggest that formal models and reasoning may be not optional but essential! Note that the most recent general collection in this area, Palanque and Paternó's 1997 book, used the Web browser as a common example—and things have got a much more complex since then. We are still awaiting the equivalent of a Model-View-Controller (MVC) or Seeheim model of interfaces in this type of environment, and similar issues to those in group editing and **Undo** may be necessary.

The other area that is fast growing in importance is the very small—multiple-context-dependent ubiquitous devices that are dynamically linking and reconfiguring themselves . . . to do something useful, one hopes! There has been relatively little work on formal issues in this area, although some colleagues and I have addressed aspects of context and location modeling (Dix, Rodden, Davies, Trevor, Friday, & Palfreyman, 2000). Clearly this is another challenge for which, again, more formal analysis will be essential.

Both global networking and ubiquity lead to emergent properties, which may mean that areas such as artificial life or the still-growing area of critical and complex systems (in the Santa Fe Institute sense) will be needed to understand or model (perhaps in a simulation sense) the situations.[4]

Architectures are also a problem for ubiquitous systems, with low-level device events needing to be marshaled and converted into higher-level events for applications. Those working in the area are talking about the need for a Seeheim-style architecture, and there has been progress with workshops dedicated to this issue (Rodden, Dix, & Abowd, 2002).

Since my earliest work in formal methods for HCI, issues of time and status phenomena have been critical (Dix 1991a, 1991b, 1996, 1998). As long as the dominant interfaces were event driven, this was largely regarded as a marginal concern. However, now we have ubiquitous devices, with physical coordinates, biomedical sensors (Allanson, 2002; Allanson & Wilson, 2002), multimedia and multimodal inputs . . . not so marginal anymore.

Although some of the work is just waiting to be reapplied, there are a number of open problems:

- nonlocalised semantics—many continuous-stream inputs such as voice or a hand gesture have meaning over a period of time, not at an instant, but there are no hard boundaries to those periods
- multiple granularity of time—large granularity time periods (e.g., today) do not map simply onto finer grain times (do we mean working day: 8 A.M.–5 P.M.; waking day: 7 A.M.–1 A.M.; etc.). There are first steps in this area (Kutar, Britton, & Nehaniv, 2001), but it is a complex issue.
- ideas of location—in the room (does in the doorway count?), close versus far (how close?)

Note that in each case we have an apparently discrete higher-level concept (in the room, today, hand gesture for “open door”), but it has “fuzzy” edges when mapped onto lower-level features.

Perhaps this sums up the problem and the challenge of formal methods in HCI. Whenever we capture the complexity of the real world in formal structures, whether language, social structures, or computer systems, we are creating

[4] Santa Fe Institute, *www.Santefe.edu*.

discrete tokens for continuous and fluid phenomena. In so doing, we are bound to have difficulty. However, it is only in doing these things that we can come to understand to have valid discourse, and to design.

14.7 FURTHER READING

There are only a few full texts explicitly on formal methods in HCI.

My own monograph covers the PIE model and many extensions and other models, including those on which status/event analysis is based.

- Dix, A. J. (1991). *Formal methods for interactive systems.* London: Academic Press.

The Table of Contents and some chapters are available online at *www.hiraeth.com/books/formal/*

See also my coauthored textbook, which has a detailed review of formal dialogue notations in Chapter 8 and other formal models in Chapter 9:

- Dix, A., Finlay, J., Abowd, G., & R. Beale (1998). *Human-computer interaction* (*2nd ed*). Hemel Hempstead: Prentice Hall.

Harrison and Thimbleby's 1990 collection includes contributions from many of those working in this area at the time.

- Harrison, M.D., & Thimbleby, H.W. (eds.) (1990). *Formal methods in human computer Interaction.* Cambridge, UK: Cambridge University Press.

Palanque and Paternó edited a more recent collection, confusingly with the same name as the above! This collection is thematic, with the contributors using their various techniques to address the Web browser as a common example.

- Palanque, P., & Paternó, F. (eds.) (1997). *Formal methods in human computer interaction.* London: Springer-Verlag.

Paternó has also produced a monograph looking at the whole design lifecycle and using ConcurTaskTrees as a central representation.

- Paternó, F. (2000). Model-based design and evaluation of interactive applications. London: Springer-Verlag.

Table of Contents and downloads are available online at: *giove.cnuce.cnr.it/~fabio/mbde.html*

For an up-to-date view of the field, probably the best source is the DSVIS conference series:

- Design, Verification and Specification of Interactive Systems, Springer-Verlag: Vienna & Berlin (1995–2002).

My formal methods in HCI Web pages include a bibliography, Web links, and further resources:

www.hcibook.com/alan/topics/formal/

Also see the Web page for this chapter, which includes links to online copies of referenced papers and further information referred to in the text:

www.hcibook.com/alan//papers/theory-formal-2003/

S
R
L

References

AD92 G. D. Abowd and A. J. Dix (1992). Giving undo attention. Interacting with Computers, 4(3): 317-342.

AWM95 Abowd, G., H. Wang, and A. Monk. A formal technique for automated dialogue development. in Proceedings of Designing Interactive Systems – DIS'95. 1995. ACM Press. p. 219–226.

A02 J. Allanson (2002). Electrophysiological Interactive Computer Systems. IEEE Computer, March 2002. pp. 60–65.

AW02 J. Allanson and G.M. Wilson (2002). Physiological Computing. Proceedings of CHI 2002 workshop on Physiological Computing, J. Allanson and G.M. Wilson (eds.), Lancaster University, UK. available online http://www.physiologicalcomputing.net/

BMDD00 P. Barnard, J. May, D. Duke and D. Duce. Systems, Interactions, and Macrotheory. ACM Transactions on Computer-Human Interaction, Vol. 7, No. 2, June 2000, Pages 222–262.

BBFMR94 Benford, S., J. Bowers, L. Fahlen, J. Mariani, T. Rodden (1994). Supporting Cooperative Work in Virtual Environments. *The Computer Journal*, **37**(8):635–668.

BCG82 Berlekamp, E.R., J.H. Conway, and R.K. Guy. *Winning ways for your mathematical plays*, volume 2: Games in particular. Academic Press, New York, 1982.

CMN80 S. K. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance with interactive systems. Communications of the ACM, 23:396-410, 1980.

CMN83 S. K. Card, T. P. Moran, and A. Newell. The Psychology of Human Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

CD95 R. Choudhary and P. Dewan (1995). A general multi-user undo/redo model. Proceedings of ECSCW'95, Stockholm, Sweden, H. Marmolin, Y. Sundblad and K. Schmidt (eds), pp. 231-246. Kluwer Acadamic.

CW96 Clarke, E. M., and J. M. Wing (1996). **Formal Methods: State of the Art and Future Directions**. *ACM Computing Surveys*, Vol. 28, No. 4, December 1996. pp. 626-643.

DH95 A. Dearden and M. Harrison. (1995). Modelling interaction properties for interactive case memories. . In Paternó, F., editor, Eurographics Workshop on Design, Specification, Verification of Interactive Systems (Pisa 1994). Springer Verlag

D94 K. Devlin (1994). Situation Theory and the Design of Interactive Information Systems, Stanford University. In Rosenberg, D and Hutchison, C. (editors) Design Issues for CSCW, Springer-Verlag (1994), pp.61-87

DR85 A. J. Dix and C. Runciman (1985). Abstract models of interactive systems. People and Computers: Designing the Interface, Ed. P. J. &. S. Cook. Cambridge University Press. pp. 13-22.

D87 A. J. Dix (1987). The myth of the infinitely fast machine. People and Computers III - Proceedings of HCI'87, Eds. D. Diaper & R. Winder. Cambridge University Press. pp. 215-228.

DH89 A. J. Dix and M. D. Harrison (1989). Interactive systems design and formal development are incompatible? In The Theory and Practice of Refinement, Ed. J. McDermid. Butterworth Scientific. 12-26.

D91 Dix, A. J. (1991). Formal methods for interactive systems. Academic Press, London. ISBN 0-12-218315-0
ToC and some chapters: http://www.hiraeth.com/books/formal/

D91b Dix, A.J. *Status and events: static and dynamic properties of interactive systems*. in *Proceedings of the Eurographics Seminar: Formal Methods in Computer Graphics*. 1991. Marina di Carrara, Italy: .

D95 A. J. Dix (1995). Dynamic pointers and threads. Collaborative Computing, 1(3):191-216.

D95b A. Dix (1995). LADA — A logic for the analysis of distributed action, in Interactive Systems: Design, Specification and Verification (1st Eurographics Workshop, Bocca di Magra, Italy, June 1994), F. Paternó, Editor. 1995, Springer Verlag: p. 317-332.

DA96 Dix, A. and G. Abowd, *Modelling status and event behaviour of interactive systems.* Software Engineering Journal, 1996. **11**(6): p. 334–346.

D96 A. J. Dix (1996). Closing the Loop: modelling action, perception and information. AVI'96 - Advanced Visual Interfaces, Eds. T. Catarci, M. F. Costabile, S. Levialdi and G. Santucci. Gubbio, Italy, ACM Press. pp. 20-28.

DML97 Dix, A., R. Mancini and S. Levialdi (1997). **The cube - extending systems for undo.** *Proceedings of DSVIS'97*, Granada, Spain, Eurographics. pp 473-495.
<http:/www.hcibook.com/alan/papers/dsvis97/>

DM97 Dix, A., and R. Mancini (1997). **Specifying history and backtracking mechanisms.** In *Formal Methods in Human-Computer Interaction*, Eds. P. Palanque and F. Paternó. London, Springer-Verlag. pp. 1-24.
<http:/www.hcibook.com/alan/papers/histchap97/>

DRW98 A. Dix, D. Ramduny and J. Wilkinson (1998). **Interaction in the Large.** *Interacting with Computers - Special Issue on Temporal Aspects of Usability*, John Fabre and Steve Howard (eds). **11**(1) pp. 9-32.
http://www.hcibook.com/alan/papers/IwCtau98/
DWR98 Dix, A., J. Wilkinson and D. Ramduny (1998). Redefining Organisational Memory - artefacts, and the distribution and coordination of work. In *Understanding work and designing artefacts* (York, 21st Sept., 1998)
D98 Alan Dix (1998). **Finding Out - event discovery using status-event analysis** *Formal Aspects of Human Computer InteractionFAHCI98*, Sheffield, 5th&6th September 1998.
http://www.hcibook.com/alan/papers/fahci98/
DFAB98 Dix, A., J. Finlay, G. Abowd and R. Beale (1998). *Human–Computer Interaction* (2nd ed). Prentice Hall.
http://www.hcibook.com/
DRDTFP00 A. Dix, T. Rodden, N. Davies, J. Trevor, A. Friday, K. Palfreyman (2000). Exploiting space and location as a design framework for interactive mobile systems ACM Transactions on Computer-Human Interaction (TOCHI), 7(3), pp. 285-321, September 2000.
DRW02 A. Dix, D. Ramduny-Ellis, J. Wilkinson (2002).**Trigger Analysis - understanding broken tasks**. In *The Handbook of Task Analysis for Human-Computer Interaction.* D. Diaper & N. Stanton (eds.). Lawrence Erlbaum Associates, 2002
abstract: http://www.hcibook.com/alan/papers/triggers2002/
more about triggers: http://www.hcibook.com/alan/topics/triggers/
D02 A. Dix (2002). Managing the Ecology of Interaction. Proceedings of Tamodia 2002 - First International Workshop on Task Models and User Interface Design, Bucharest, Romania, 18-19 July 2002. <http://www.hcibook.com/alan/papers/tamodia2002/>
D02b A. Dix (2002). Towards a Ubiquitous Semantics of Interaction: phenomenology, scenarios and traces. Proceedings of DSV-IS 2002 -Design, Specification, and Verification of Interactive Systems. Rostock, Germany, June 2002 (published Springer, LNCS)
D02c A. Dix (2002). Embodied Computation.
http://www.hcibook.com/alan/topics/embodied-computation/
D02d A. Dix (2002). In Praise of Randomness.
http://www.hiraeth.com/alan/topics/random/
DMF01 Gavin J. Doherty, Mieke Massink and Giorgio Faconti, **Using Hybrid Automata to Support Human Factors Analysis in a Critical System**, in Formal Methods in System Design, 19(2), 143–164, 2001
EGP94 Ellis, G.P., Finlay, J.E. and Pollitt, A.S. HIBROWSE for Hotels: bridging the gap between user and system views of a database. Proc. IDS '94 2nd Int'l Workshop on User Interfaces to Databases, Lancaster, UK, April 1994. Springer Verlag: Workshops in Computer Science. pp. 45–58
EG89 C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. SIGMOD Record, 18(2):399{407, June 1989. 1989 ACM SIGMOD International Conference on Management of Data.
EW94 Ellis, C., and Wainer, J. (1994). Goal based model of collaboration. Collaborative Computing, 1, 1.
ES95 T. Elwert and E. Schlungbaum (1995). Modelling and generation of graphical user interfaces in the TADEUS approach. In P. Palanque and R. Bastide (eds.) Design, Specification and Verification of Interactive Systems '96 (Proceedings of DSVIS'95. Toulouse, France, June 1995). Springer. pp. 193–208.
FD96 J. Finlay and A. Dix (1996). An Introduction to Artificial Intelligence. UCL Press / Taylor and Francis, ISBN 1-85728-399-6.
FS95 J. Foley and P. Sukaviriya (1995). History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In Paternó, F., editor, Eurographics Workshop on Design, Specification, Verification of Interactive Systems. Springer Verlag. pp. 3–14.
G70 M. Gardner. The fantastic combinations of John Conway's new solitaire game ``life''. *Scientific American*, pages 120–123, October 1970.
GJA92 W. D. Gray, B. E. John, and M. E. Atwood. The precis of project ernestine or an overview of a validation of goms. In P. Bauersfeld, J. Bennett and G. Lynch, editors, Striking a Balance, Proceedings of the CHI'92 Conference on Human Factors in Computing Systems, pages 307-312. ACM Press, 1992.
G93 Grossman, R.L., *et al.*, ed. *Hybrid Systems*. 1993, LNCS 736, Springer Verlag.
Harrison, M.D. and Thimbleby, H.W., editors (1990). Formal Methods in Human Computer Interaction. Cambridge: Cambridge University Press.
H85 C. A. R. Hoare. "Communicating Sequential Process." London: Prentice-Hall International, 1985
H79 Hofstadter, D.R. (1979). **Gödel, Escher, Bach: an Eternal Golden Braid**. Basic Books

ISO89 ISO (1989), Information Processing Systems, Open Systems Interconnection, LOTOS --- A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807, Geneva
K67 C. Kilmister (1967). Language, Logic and Mathematics. English Universities Press, London.
K97 Kotze, P. (1997). The Use of Formal Models in the Design of Interactive Authoring Support Environments. DPhil Thesis. University of York, York, UK. YCST 97/09
KBN00 M. Kutar, C. Britton and C. Nehaniv. Specifiying multiple time granularities in interactive systems. Palanque and Paternó (eds), *DSV-IS 2000 Interactive Systems: Design, Specification and Verification*. LNCS 1946, Springer 2001, pp. 169–190.
LC98 K. Larson and M. Czerwinski. Web Page Design: Implications of Memory, Structure and Scent for Information Retrieval, In Proceedings of CHI 98, Human Factors in Computing Systems (LA, April 21-23, 1998), ACM press, 25-32.
M97 R. Mancini (1997). Modelling Interactive Computing by exploiting the Undo. Dottorato di Ricerca in Informatica, IX-97-5, Università degli Studi di Roma "La Sapienza".
M94 S. Marsh (1994). Trust in Distributed Artificial Intelligence. In Castelfranchi and Werner (eds) Artificial Social Societies, Springer Verlag, Lecture Notes in AI number 830, 1994. pp. 94–112.
M56 G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity to process information. Psychological Review, 63(2):81-97, 1956.
M80 R. Milner. A Calculus of Communicating Systems. Lecture Notes in Computer Science 92. Springer-Verlag, 1980
NPPSB01 D. Navarre. P. Palanque, F. Paternó, C. Santoro and R. Bastide. A tool suite for integrating task and system models through scenarios. C. Johnson (ed), *DSV-IS 2001 Interactive Systems: Design, Specification and Verification*. LNCS 2220, Springer 2001, pp. 88–113.
PB95 P. Palanque and R. Bastide. Petri net based design of user-driven interfaces using the interactive cooperating objects formalism. In F. Paternó, editor, Interactive Systems: Design, Specification and Verification (1st Eurographics Workshop, Bocca di Magra, Italy, June 1994), pages 215-228. Springer-Verlag, Berlin, 1995.
PB96 P. Palanque and R. Bastide. Formal specification and verification of CSCW. In M. A. R. Kirby, A. J. Dix, and J. E. Finlay, editors, People and Computers X - Proceedings of the HCI'95 Conference, pages 213-231. Cambridge University Press, Cambridge, 1996.
PP97 Palanque, P. and Paternó, F., editors (1997). Formal Methods in Human Computer Interaction. London, Springer-Verlag.
P00 Paternó, F. (2000). Model-Based Design and Evaluation of Interactive Applications. London, Springer-Verlag.
ToC and downloads: <http://giove.cnuce.cnr.it/~fabio/mbde.html>
PS00 F. Paternó and C. Santoro. Integrating model checking and HCI tools to help designers verify user interface properties. Palanque and Paternó (eds), DSV-IS 2000 Interactive Systems: Design, Specification and Verification. LNCS 1946, Springer 2001, pp. 135–150.
PMG01 F. Paternó, G. Mori and R. Galimberti. CTTE: an environment for analysis and development of task models of cooperative applications. Proceedings of CHI'01, Vol 2, ACM Press, 2001.
P62 Petri, C. Kommunikation mit Automaten. PhD thesis, University of Bonn, Bonn, West Germany, 1962
PNW02 Petri Nets World. University of Aarhus, Denmark. accessed October 2002. <http://www.daimi.au.dk/PetriNets/>
P85 G. Pfaff and P. J. W. ten Hagen, editors. Seeheim Workshop on User Interface Management Systems. Springer-Verlag, Berlin, 1985.
PK92 A. Prakash and M. J. Knister (1992). Undoing actions in collaborative work. Proceedings of CSCW'92. Toronto Canada. pp. 273-280, ACM Press.
PK94 A. Prakash and M. J. Knister (1994). A framework for undoing actions in collaborative systems. ACM Transactions on Computer Human Interaction, 1(4):295-330.
R81 P. Reisner. Formal grammar and human factors design of an interactive graphics system. IEEE Transactions on Software Engineering, SE-7(2):229-240, 1981.
RNG96 M. Ressel, D. Nitsche-Ruhland and R. Gunzenhfiuser (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. Proceedings of CSCW'96. Boston USA. ACM Press pp. 288-297
RG99 M. Ressel and R. Gunzenhfiuser (1999). Reducing the problems of group undo. Proceedings of Group'99. Phoenix,USA. ACM Press, pp. 131-139
RS96 C. Roast and J. Siddiqi (1996) Formally assessing software modifiability. In C. R. Roast and J. Siddiqi, editors. BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996, Electronic Workshops in Computing. Springer-Verlag, 1996.

R96 Rodden, T., (1996) Populating the application: a model of awareness for cooperative applications, in: CSCW '96. *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work*, pp 87–96.

RDA02 T. Rodden, A. Dix and G. Abowd (2002). Concepts and Models for Ubiquitous Computing, Workkshop at UbioComp 2002, Göteborg, 29th Sept 2002 http://www.hcibook.com/alan/conf/ubicomp-models/

R08 W. Rouse Ball (1908). A Short Account of the History of Mathematics (fourth edition). Dover, New York.

SBB97 Sandor, O., Bogdan, C., Bowers, J, (1997), Aether: An Awareness Engine for CSCW, in J.Hughes et al (eds.) Proceedings of ECSCW'97: the Fifth European Conference on Computer Supported Cooperative Work, pp 221–236, Kluwer Academic Press.

S95 Shepherd, A., *Task analysis as a framework for examining HCI tasks*, in *Perspectives on HCI: Diverse Approaches*, A. Monk and N. Gilbert, Editors. 1995, Academic Press: London. p. 145–174.

S84 Shneiderman, B., Response time and display rate in human performance with computers, ACM computing surveys, Vol. 16, No. 3, 265-286, Sept 1984.

S88 J. M. Spivey. The Z Notation: A Reference Manual. Prentice Hall International, Hemel Hempstead, 1988.

SEP02 Entry on **The Church-Turing Thesis** in Stanford Encyclopedia of Philosophy. accessed October 2002. <http://plato.stanford.edu/entries/church-turing/>

S87 Suchman, L. (1987). Plans and Situated Actions: The problem of human–machine communication. Cambridge University Press.

S82 B. Sufrin. Formal specification of a display editor. Science of Computer Programming, 1:157-202, 1982.

SJZYC98 C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems.ACM Transactions on Computer Human Interaction, 5(1):63-108.

S00 C. Sun (2000). Undo any operation at any time in group editors. Proceedings of CSCW'2000, Philadelphia, PA USA, pp. 191-200, ACM Press. in ACM Digital Library

SSCMS96 P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy and E.Salcher (1996). Declarative interface models for user interface construction tools: the MASTERMIND approach. In L. Bass & C. Unger (eds.) Engineering for Human–Computer Interaction. Proceedings of the IFIP WG2.7 working conference. Yellowstone Park, August 1995. Chapman & Hall, London. pp. 120-150.

TCT TACIT: Theory and Applications of Continuous Interaction Techniques, EU TMR Network ERB FMRX CT97 0133. http://kazan.cnuce.cnr.it/TACIT/TACIThome.html

TCJ01 Thimbleby, H., P. Cairns, and M. Jones (2001). Usability Analysis with Markov Models. ACM Transactions on Computer-Human Interaction, Vol. 8, No. 2, June 2001, Pages 99–132.

TGGCR96 J. Torres, M. Gea, F. Gutierrez, M. Carbrera and M. Rodriguez (1996). GRAPLA: and algebraic specificatiion language for interactive graphic systems. In F. Bodart and J. Vanderdonckt. Design, Specification and Verification of Interactive Systems '96 (Proceedings of DSVIS'96. Namur, Belgium, June 1996). Springer. pp. 272–291.

TSDS95 L. Tweedie, R. Spence, H. Dawkes and H. Su. The Influence Explorer. Companion Proceedings CHI '95. ACM Press, 1995, 129-130

vN56 von Neumann, J., (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. Shannon and J McCarthy, editors, Automata Studies. Princeton University Press. 1956

vN66 von Neumann, J., (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966. Edited and completed by A.W. Burks

W97 Wegner, P. (1997), Why Interaction is More Powerful Than Algorithms, *CACM*, **40**(5):80–91, May 1997.

W99 C.A. Wûthrich. An analysis and model of 3D interaction methods and devices for virtual reality. D.J. Duke and A. Puerta (eds). *DSV-IS 1999 Design, Specification and Verification of Interactive Systems*. Springer 1999, pp. 18–29.

WF86 Winograd, T. and F. Flores (1986). Understanding computers and cognition : a new foundation for design. Addison-Wesley.