ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/232062764

Syntonicity and the psychology of programming

Conference Paper · January 1998

CITATIONS
20

READS
985

1 author:

Stuart Watt
Turalt Inc.
93 PUBLICATIONS **824** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Open Mentor View project

Design eBook User Interface to Support Reading for Comprehension View project

All content following this page was uploaded by Stuart Watt on 03 June 2014.

The user has requested enhancement of the downloaded file.

Syntonicity and the psychology of programming

Stuart Watt
Knowledge Media Institute and Department of Psychology
Open University
Walton Hall, Milton Keynes, MK7 6AA, UK
+44 1908 653169
S.N.K.Watt@open.ac.uk

Abstract. This paper argues that the psychology of programming is at least partly 'psychological' in character, rather than being purely physical or computational. That is, when people are learning a programming language, or an execution model for a programming language, they think about them almost as homunculi — agents which act according to beliefs, desires, and intentions. At the moment, this is just a hypothesis — although it is potentially an important one to the psychology of programming. In this paper, I will try to make this hypothesis a bit clearer, point out some of the ways that it can be investigated, and discuss some of its further implications.

Keywords. Syntonicity, theory of mind, teaching and learning programming, psychology of programming.

Introduction

In this paper I will argue that syntonicity, a concept borrowed from Papert's Logo (1993) but originally due to Freud, should be once again taken seriously as an important concept in the psychology of programming. The argument is based on research in developmental psychology which suggests that, even in very young children, there are essentially separate faculties for physical and psychological reasoning (Carey, 1985; Wellman, 1990). Even young children have a 'theory of mind' — a kind of 'common-sense' psychology — which allows them to ascribe mental states to others, and to use those ascribed mental states to predict and understand other people's behaviour. Like Piaget's theories of the development of reasoning, these theories suggest that there may be several stages through which children pass before they develop the full theory of mind that adults have. Wellman (1990), for example, suggests that before developing a complete theory of mind, children have a "simple desire psychology", meaning that they can ascribe goals, wants, and intentions to others before they can ascribe beliefs. That is, children can ascribe motivational mental states before they can ascribe informational mental states.

A key finding in the theory of mind research is that there seems to be a significant developmental shift in children approximately between the ages of two and four

(Wellman, 1990). This shift is characterised by a change in childrens' thinking so they begin to be able to distinguish between their own and other peoples' mental states. In particular, this means a shift away from something like Piagetian egocentricity to an ability to take other people's roles, to being able to 'put themselves in another's shoes' (Astington & Gopnik, 1991; Perner, 1991). It is this that links theory of mind and syntonicity, because syntonicity in programming means, basically, being able to put oneself in the *program's*, or the *programming language's*, shoes.

But what does this mean for the psychology of programming? Syntonicity, as Papert uses it, represents a 'resonance' between external forms and concepts and what people know about themselves. Papert describes Logo's 'turtles' as providing a target for syntonicity. "Children can *identify* with the Turtle and are thus able to bring their knowledge about their bodies and how they move into the work of learning formal geometry" (Papert, 1993, original emphasis). The word 'identify' here is a big clue: being able to identify with people is an important sign of a mature theory of mind (Meltzoff & Gopnik, 1993).

The hypothesis is simply this: syntonicity is important because it extends the scope of identification to programs, to programming languages, and to execution models. We begin to think about programs as psychological entities, rather than physical ones. After all, programs do things — they behave as if they have goals, the very components of Wellman's simple desire psychology. Syntonic programming should make it easier for people to understand how and why programs do what they do, because people can identify with those programs, and see things from their point of view.

If this 'syntonicity hypothesis' is true, it offers a new perspective on people's grasp of programs, programming languages, and execution models, and a way for us to look at the different stories we tell about a language, not only at the informational level, but at a psychological level. We can look at the different stories with a theory of when some are easier to 'grasp', and why. In the future, theories such as this may help us to design programming languages so that they are easier to learn.

Background: syntonicity and programming

It might seem strange that we think about computers in psychological terms rather than computational ones, but for those who doubt that social and psychological phenomena can extend to computers, there is good evidence in the experiments of Nass, Steuer, and Tauber (1994). The importance of human social and psychological cues is also a founding principle of research on the theatre metaphor (e.g. Laurel, 1990) and the social interface (e.g. Ball *et al.*, 1996).

There is another piece of evidence that we often think about computers psychologically. The very design of computers was, in fact, based on human psychology. "Von Neumann and Goldstine were *not* inventing the computer. They already knew what a computer was, what a computer did, and how long it took a computer to do it" (Bailey, 1992, original emphasis). Today's electronic computers are designed in the image of the *human*

computers that preceded them. It is not altogether surprising, then, that we think about computers as if they are people. Put simply, computers aren't computational as much as social and psychological, as people think about them. "The computer is a new kind of thing — psychological yet a thing" (Turkle, 1984).

Enough of the hardware. People identify with programs and operating systems (Laurel, 1990) as well as grey boxes. There are many different ways that one can identify with the kind of elements commonly found in computing, and these are subtly different in their operation and effects. Among others, all the following different kinds of identification are possible.

- Identifying with the computer
- Identifying with the operating system
- Identifying with the interpreter
- Identifying with the compiler
- Identifying with the language
- Identifying with a program statement or expression (in imperative programming)
- Identifying with a program procedure or function (in imperative programming)
- Identifying with an object (in object-oriented programming)
- Identifying with a clause (in declarative programming)
- Identifying with a process (in data flow programming)

From this, it is clear that there are different kinds of syntonicity. First, there is a kind of general syntonicity, which is independent of the programming language or paradigm involved. At this level, people identify with the physical or functional processes involved in writing or running a program. However, there is a second level of specific syntonicity which can vary from one paradigm to another. In this, the units which go to make up the language, whatever they may be, provide different targets for identification.

Another useful point, made by Papert (1993), distinguishes between "body syntonicity" (where the identification relates principally to the body, and is physical) and "ego syntonicity" (where the identification relates principally to the mind, and is mental). Logo, at least when used to teach geometry, explores body syntonicity most — ego syntonicity requires language elements which correspond to the turtle's having mental states; beliefs, desires, and intentions.

One of the easiest ways to study the extent to which syntonicity plays a role in people's grasping of programming languages is to look at anthropomorphism. By anthropomorphism I do not mean the 'jokey' kind of anthropomorphism of Lewis Carroll or Rev. W. D. Awdry — 'Thomas the Tank Engine' anthropomorphism — I simply mean the use of mentalistic descriptions for things which are not people's minds. In the field of programming, anthropomorphism is significant when people begin to use mentalistic descriptions for the objects involved in developing and running programs, for example, computers, interpreters, compilers, procedures, and programs (Laurel, 1990; Nass *et al.*, 1994).

Anthropomorphism has recently become a topic of interest in the human-computer interaction community, mainly through the research on the 'theatre metaphor' (Laurel, 1990) and the "social interface" (Ball *et al.*, 1996). Anthropomorphism, however, is not universally accepted as a Good Thing; a lot of the time it just seems to get in the way — people don't want to have to negotiate their tasks with "some little dip in a bow tie" (Laurel, 1990). The point is that it seems unavoidable, to some extent at least. People simply act anthropomorphically to computers anyway; as Nass *et al.* (1994) show, "computers are gendered social actors" — even to experienced computer users.

Now the point I am trying to make — or, rather, the hypothesis I am suggesting that should be seriously considered — is simply this: in looking at the psychology of programming we borrowed superficial ideas from Logo, and largely neglected the deep ideas derived from the use of syntonicity at its heart. So let's be clear about this. This possibility of the importance of syntonicity to the psychology of programming — ego syntonicity as well as body syntonicity — leads to one major hypothesis and two corollaries. I won't spend much time discussing these here; rather, I am just making them explicit here so they are flagged for future empirical study:

The Syntonicity Hypothesis. Programs and execution models are at least partly psychological, rather than being purely computational or physical. That is; people think about the behaviour of a program in mentalistic terms, rather than formal, logical, mathematical, or physical ones.

Corollary 1. The more syntonic the programming language, the easier it will be to learn.

Corollary 2. Execution models will be easier to grasp if people can identify with them.

A brief digression: syntonicity and metaphor

It is worth pointing out here that syntonicity and metaphor are not quite the same thing, but they are related. If we return to Papert's use of body syntonicity in Logo, the idea is that children can use their own bodies as if they were the turtle. If they cannot grasp the procedure to draw a circle, they can simply act it out by *being* the turtle. In one sense, this is a metaphor, but it is a very special kind of metaphor. It is, in Johnson's (1987) terms, "image-schematic" — with a basis in experience. So, from the point of view of syntonicity, all metaphors are not equal. In a sense, *any* story we tell about a programming language's execution (or 'virtual machine') is a metaphor, as are most ways of thinking about any physical machine. The question is: are syntonic stories easier to grasp than dissociated ones. ('Dissociated' is, generally speaking, the opposite of 'syntonic'.)

So far, I have said very little about the role of metaphor in the psychology of programming — explicitly, at least. There is, however, a deep connection between the syntonicity hypothesis and the role of metaphor. Metaphor is not a superficial thing that can be *added* to a programming language or execution model, it is something that is right at the heart of the way that people grasp it, that shapes how people identify with that programming

language.

Let me try and make this clear. I am suggesting that a language cannot be understood except through one metaphor or another — but that metaphors are not all the same. It is simply that some metaphors are more useful than others, depending on the task in hand, and on the individual people who are trying to grasp the programming language concerned. Metaphors that people can identify with — syntonic metaphors if you like — seem to be better at getting the point across than dissociated metaphors (I'll show this in the next section, by comparing two different stories of Prolog execution). When designing a metaphor or an execution model, we should look at how easy it might be to identify with that model, and for this the psychology of ego syntonicity may turn out to be just as important as body syntonicity.

If syntonicity, and particularly ego syntonicity, works in the way I suggest in this paper, then there are significant implications for the psychology of programming. Not only do we have some clues about whether some stories are better than others, we have some clues about which stories are better, and why. We can begin to teach programming, and to design programming languages and execution models, so that we go with the grain of syntonicity, rather than against it. Perhaps we should finally abandon the idea that Turing-like notions of computational equivalence have any real significance for learning programming; perhaps we should start designing programming languages and execution models simply so that they are easier to grasp.

Case studies

Given this theory of syntonicity and its effects in the psychology of programming, I'll now look at a few case studies to show how syntonicity can have a definite, and not always positive, effect on people's grasp of a few programming languages.

In the first case study, I'll look at a single language (Prolog) and show how different stories can fundamentally influence the syntonic effects. That is, syntonicity is *not* directly a property of the syntax, or even the semantics or pragmatics of a language, but a property of the stories that we tell about it.

In the second study, I'll take a rather different tack. This time, I'll look at some of the developments that have followed in the footsteps of Logo, and how syntonicity has changed throughout these developments. In particular, I'll look at their actual ease of use, and compare this to the different kinds of syntonic effect within the different languages.

Case study 1: Prolog

Prolog is not, at least at first glance, the best language to look at if we want to know how people learn to program. After all, it works in a radically different way from most programming languages, and normal mortals often find it hard to grasp. But there are

several reasons for looking at Prolog in detail. First, while the behaviour of the language is fixed, a few different execution stories (Pain & Bundy, 1987) and many different tracers have been developed for it, and these have been studied empirically in some detail (Mulholland, 1995a). This means that we can look at the effects of these different tracers and execution models without worrying too much about the underlying language. Secondly, the language itself is conceptually pretty simple — it doesn't introduce many concepts, although those that it does can sometimes be pretty hard to grasp. Indeed, there is an element of myth about the difficulty of Prolog — often the people who find it hardest are those who are most expert with other programming languages, so much of the difficulty may simply be due to a negative transference effect. Thirdly, Prolog is abstract — it provides a completely sealed virtual machine, so that people learning it do not have to worry about aspects of the underlying computer's hardware or operating system — as is the case with some languages, C and Pascal for instance.

In fact, there are several common execution models for Prolog (Pain & Bundy, 1987), including the AND/OR tree model, the OR tree model, the Byrd box model, and the choice-point model. Of these, we'll only look at the Byrd box model and the choice-point model in more detail, as the contrast between them most clearly shows the effect of syntonicity.

In the Byrd box model, each procedure (or Prolog predicate) is conceived as a box, with inputs and outputs, rather as drawn in figure 1. The important point to note about the Byrd box model is that "the 'box' idea delineates the procedure as the prime focus of attention" (Byrd, 1980) — that is, each box, and therefore each procedure, has its own control flow; a control that is separate from the control of the other boxes and the other procedures. In the Byrd box model, you are encouraged to identify with the procedure, not with the interpreter as a whole. In the anthropomorphic limit, procedures may even become characters. One overview of Logo (Bundy, 1978) says: "It is sometimes useful to think of each call of a procedure as a 'little man'. Arguments to the little man go in through his ears. Results come from his mouth. Other things he does, like effects, are achieved by other organs." And there are even drawings to match!

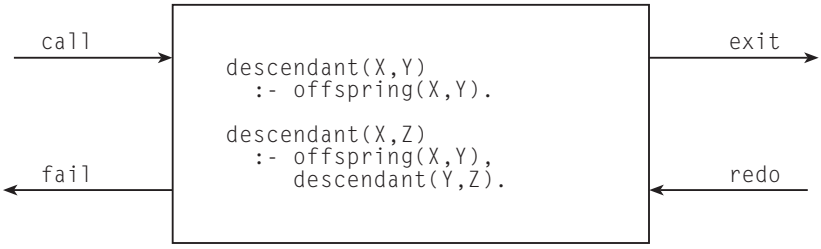

Figure 1. A Byrd box model for the procedure 'descendant' (after Byrd, 1980)

The choice-point model, on the other hand, is more syntonic. Mulholland (1995b) illustrates this model using one variant of the myth of Theseus and the Minotaur. According to this myth, as Theseus passed through the maze, he left coins to mark which passages he had followed. To retrace his steps, all he had to do was follow the coins back. This is a syntonic metaphor for Prolog — Theseus corresponds to the Prolog interpreter;

when the interpreter backtracks it goes back to the previous marked choice-point.

According to the main hypothesis, then, the choice-point model (where people are encouraged to identify with one thing, Prolog) should be easier to teach than the Byrd box model (where people are encouraged to identify with each procedure call separately). This effect will be magnified when a syntonic metaphor is used in the teaching process. Sure enough, Mulholland has found that choice-point models do help to make it easier to understand Prolog execution traces.[1]

Case study 2: Logo, StarLogo, Playground, and Cocoa

Even though they have similar audiences, Logo and Cocoa[2] are radically different in their aims and in the way that they work underneath. Basically, Logo was intended to help children learn about mathematics — Cocoa , and its predecessors Playground (Fenton & Beck, 1989) and KidSim (Cypher & Canfield Smith, 1995) were more simulation environments; they were far more frequently used to study ecosystems and things of that sort.

As far as their execution models are concerned, the main difference between Logo and Cocoa/KidSim is that where Logo is procedural, Cocoa is declarative. That is, to write a Logo program, a child has to write down a procedure which tells the turtle what to do. This is where syntonicity comes in — it allows the child to study these procedures by simply acting them out for themselves. Syntonicity gives them a way to identify with the turtle, and so to understand how these programs work. Cocoa, on the other hand, is declarative — to write a Cocoa program the child writes a set of rules which deal with particular situations, and when these rules are put together they tell the objects how to behave.

This difference is an important one. Commonly, we see the difference between procedural and declarative knowledge as a difference of whether we make the execution story explicit or implicit. But in practice, the difference between the two types of knowledge runs far deeper than that. The way that people learn declarative knowledge (such as facts and rules) is simply different from the way that people learn procedural

[1] This argument is perhaps unfair to Mulholland. He has other reasons for suggesting that the choice-point model is better, for novices at least, than the Byrd box model. And since he has evaluated them empirically in very substantial detail — he's very likely right (Mulholland, 1995a). But this doesn't necessarily invalidate the hypothesis — the syntonic nature of 'Mulholland's metaphor' could still be adding to the strength of the choice-point model. One way to evaluate this might be to look at the protocols Mulholland used, and to look for a significant difference in the use of intentional terms to describe the Prolog interpreter. If the choice-point model is truly more syntonic, it is very likely that this would have showed up in the protocols. This is planned for future work.

[2] Cocoa is available from http://www.cocoa.apple.com/. For the purposes of this paper, it is enough to think about it as a cleaned-up version of KidSim (Cypher & Canfield Smith, 1995).

knowledge (such as skills).[3]

The point is that just as in the case of Prolog, these two different stories are equivalent, at least computationally. What has yet to be proven is whether they are *psychologically* equivalent — and the hypothesis in this working paper is that computationally equivalent models may differ radically in their syntonicity, and, therefore, in their psychological appeal.

It is also worth thinking about StarLogo (Resnick, 1994) in this context. StarLogo, is in many ways, a 'next generation' Logo, and it has much in common with Playground. Although it was designed mainly to look at the effects of centralised thinking, StarLogo adds a few features that do make its turtles a bit more syntonic than Logo's. First, StarLogo clarifies the relationship between the turtles and their environment; it adds an active model of the environment as an array of 'patches' which evolve over time, much as turtles do themselves. This environment model is no longer passive; turtles can change the environment and the state of a patch can influence the behaviour of a turtle nearby. Secondly, in StarLogo turtles have better 'senses' through which they can perceive the environment around them. And third, like Playground, StarLogo adds 'demons', procedures which run constantly — demons are, perhaps, most interesting because they make goals and beliefs explicit in exactly the way needed for ego syntonicity. For example, here's one of Resnick's rules from a model of ant colony foraging behaviour.

```
to find-food-demon
if (not carrying-food?)
    and ask patch-here [food > 0]
  [set-carrying-food? true
   ask-patch-here [set-food food - 1]
   set-drop-size 35
   right 180 forward 1]
end
```

This demon represents an ant's 'want' for food, implementing a strategy for achieving the goal of finding food; it is conditional on the ant's 'believing' that it isn't carrying food. In fact, in Resnick's model the ants have four demons, and all of them correspond to an ant's goals under different circumstances. This is quite ego syntonic — these demons

[3] I ought to be a bit clearer about this. The difference between procedural and declarative knowledge bears some similarity to Anderson's theory of the acquisition of skill, but there are important differences. For example, there are clearly procedural elements even in the rules used in Anderson's (1982) model of multiple column addition — subgoals, for example, have a definite order to them. The area where I believe syntonicity is most important are the earliest stage in the learning process — acquiring the declarative encoding. Anderson is rather coy about how declarative knowledge is acquired for the earliest encoding — it seems to rest mainly on how "the learner receives instruction and information about a skill" (Anderson, 1982) — as well as background common sense and general heuristics. It would seem probable that there is a lot more to it than that — imitation, for example, seems to play an important role. This, once again, would imply that theory of mind plays a bigger role than might be apparent at first glance. Imitation is important to theory of mind (Meltzoff & Gopnik, 1993), and probably also to skill acquisition. If syntonicity is important to skill acquisition — and for some skills this certainly seems to be the case — then there is a whole can of worms involved in the psychology of skill acquisition. I'd rather not open this can now; here I should just be taken as clearly labelling it 'worms inside'.

correspond to behaviour like 'what to do when you find food'. On the other hand, the KidSim/Cocoa equivalent for this is no longer ego syntonic, because now what we see is only the 'objective' physical adjacency of, say, an animal and some food. By using graphical rules and showing only a 'God's eye' view of the world, paradoxically, the view from 'inside' the animal has been lost.

So, according to this hypothesis, KidSim/Cocoa could actually be harder to learn than Logo/StarLogo/Playground, not easier. This is because the several rules that may go to make up a single action (climbing over an obstacle, for example) are not connected, but are driven entirely by several different situation action rules — action rules that do not correspond syntonically to a complete goal or action. Because these rules are dissociated, quite simply, there is nothing there in the execution model to identify with (except, perhaps, God). This loss of syntonicity may make some kinds of modelling much harder with Cocoa than with Logo. Again, this is an area where more empirical evaluation is needed — not a simple leap of faith in the 'graphics good, text bad' mould.

Empirical studies of syntonicity: future work needed

So far, there have not been any clear empirical studies to show the benefits, or otherwise, of syntonicity. Evaluations of Logo's successors (e.g. Gilmore, Pheasey, Underwood, & Underwood, 1995) have generally not been comparative. This means, for example, there is no empirical evidence whether KidSim/Cocoa's graphical (but dissociated) rules are easier or harder to learn than Playground's textual (but syntonic) rules. Indeed, it is hard to devise experiments which allow the syntonicity and the modality to be evaluated separately. One way to test this may simply be to provide a hint condition for people building a simple simulation, explicitly suggesting that they try to identify with the characters concerned, and to see whether this makes their programming easier or harder.

These details aside, there is evidence that anthropomorphic characters — agents one can identify with syntonically — can assist people's problem solving in educational software (Lester, Converse, Stone, Kahler, & Barlow, 1997). There is, therefore, some evidence that syntonicity might have an effect, and none that that it doesn't. At this point, then, the syntonicity hypothesis cannot be rejected, although further studies are certainly required.

'So what?'

At might, at first glance, seem that something as arcane as syntonicity doesn't have an important role to play in the psychology of programming. Why should we bother with it? There are several replies to this. First, syntonicity already has made a significant contribution to the psychology of programming. It was central to the early developments in Logo, but its importance has been rather overshadowed by the more recent versions and derivatives of the language. Even so, Papert (1993) was making very strong claims about the importance of syntonicity.

A second reply is simply that syntonicity is endemic to all languages and paradigms. That is, the way that people think about programs is in mentalistic and psychological terms, and there is no way that it can be 'omitted'. This is because syntonicity is a reflection of people's common sense psychology being extended to a new kind of artifact — a program — and because common sense psychology is so deeply embedded in people's thinking, it cannot simply be removed or ignored. The best that we can do is to recognise it and live with it.

A third reply is that we need to be very careful about our new languages' design goals. It is easy — too easy — to design a new programming language. It is important to ask (at least) one question before doing this — who, if anybody, is going to use this new language, and why.

Syntonicity offers a chance for a new theory of 'graspability', of what makes a language easy or hard to predict. This theory of graspability is not set in computational terms but in (common-sense) psychological ones. The intention is that for complete novices, without any background in computation, a mathematical or computational description of a language is relatively useless, because it simply moves the problem of graspability to the mathematical or computational description. Instead, syntonicity offers a way of casting graspability in people's common-sense prediction of how people behave.

Conclusions: syntonicity, psychology, and skills

In this paper, I have suggested that programs are often psychological entities rather than merely physical or computational ones. People treat programs as if they have mental states. Sometimes, of course, this doesn't work particularly well, and it can become hard to understand why the program is doing what it is doing. In fact, it is only really easy to understand programs when we can identify with them, when we can put ourselves inside the program.

If this is true — and that is yet to be proven conclusively — then there are some important ramifications. If we want to design languages that are easy to learn, we need to design them in such a way that we make it easy for people to identify with its programs. While Logo was designed with this goal explicitly — and it certainly seems that Logo did make it significantly easier for some people to learn to program compared to most contemporary languages — there is still a lot of room for improvement. For example, although Logo is good at being body syntonic (the turtle is easy to identify with physically), it doesn't seem to be so good at being ego syntonic. Psychological identification with the beliefs, goals, and desires of a Logo program are rather harder than physical identification with the turtle. There is nothing obviously like a set of beliefs or desires inside a turtle — and although we could simply add them, we need to connect them to the world in the right way. Beliefs and desires are not interesting except to the extent that they cause behaviour — a turtle's beliefs would need to cause its behaviour in the way that we normally expect according to our common-sense psychology, if it is to be properly ego syntonic.

Perhaps in learning to program, and in learning other skills, imitation plays an important role, we copy other peoples' — and other objects' — behaviour when we learn a skill. But because some things are easier to imitate than others, it matters what kind of example we provide in the learning process. Over and above the examples that we use to teach, the way that we show the processes involved in the examples matters. Learning a skill doesn't simply start with passively receiving instructions, it starts with actively looking at others using their skill, and, in time and through identifying with them, we gradually learn to understand how and why their skill works.

If this is true, we need to ensure that we take it into account in the psychology of programming. Models of skill acquisition and programming language acquisition which ignore aspects of syntonicity may need to be reconsidered in this light. Syntonicity is not the only factor which needs to be taken into account when devising an execution model, a programming metaphor, or a story; but even so, it certainly appears to be a factor, and possibly an important one.

Acknowledgements

Thanks to Chris McKillop and Paul Mulholland for many helpful thoughts on the ideas in this paper. Both of them achieved the near impossible, and persuaded me to actually put them down in written form.

References

Anderson, J. R. (1982). Acquisition of Cognitive Skill. *Psychological Review, 89*, 369-406.

Astington, J. W., & Gopnik, A. (1991). Theoretical Explanations of Children's Understanding of the Mind. *British Journal of Developmental Psychology, 9*, 7-31.

Bailey, J. (1992). First We Reshape Our Computers, Then Our Computers Reshape Us: The Broader Intellectual Impact of Parallelism. *Daedalus, 121*(1), 67-86.

Ball, G., Ling, D., Kurlander, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Van Dantzich, M., & Wax, T. (1996). Lifelike Computer Characters: The Persona Project at Microsoft Research. In J. M. Bradshaw (ed.), *Software Agents*. Cambridge, Massachussetts: AAAI Press/MIT Press.

Bundy, A. (ed.). (1978). *Artificial Intelligence: An Introductory Course*. Edinburgh: Edinburgh University Press.

Byrd, L. (1980). *Understanding the Control Flow of Prolog Programs*. Paper presented at the Logic Programming Workshop, Debrecen, Hungary.

Carey, S. (1985). *Conceptual Change in Childhood*. Cambridge, Massachusetts: MIT Press.

Cypher, A., & Canfield Smith, D. (1995). *KidSim: End User Programming of Simulations*. Paper presented at the CHI'95 conference, Denver, Colorado.

Fenton, J., & Beck, K. (1989). *Playground: An Object-Oriented Simulation System for Children of All Ages*. Paper presented at the OOPSLA'89, New York.

Gilmore, D. J., Pheasey, K., Underwood, J., & Underwood, G. (1995). *Learning Graphical Programming: An Evaluation of KidSim*. Paper presented at the Human-Computer Interaction: Interact'95.

Johnson, M. (1987). *The Body in the Mind: The Bodily Basis of Meaning, Imagination, and Cognition*. Chicago: University of Chicago Press.

Laurel, B. (1990). Interface Agents: Metaphors With Character. In B. Laurel & S. J. Mountford (eds.), *The Art of Human-Computer Interface Design* (pp. 355-365). Addison-Wesley.

Lester, J. C., Converse, S. A., Stone, B. A., Kahler, S. E., & Barlow, S. T. (1997). Animated Pedagogical Agents and Problem-Solving Effectiveness: A Large-Scale Empirical Evaluation. In B. du Boulay & R. Mizoguchi (eds.), *Artificial Intelligence in Education* (pp. 23-30). IOS Press.

Meltzoff, A., & Gopnik, A. (1993). The Role of Imitation in Understanding Persons and Developing a Theory of Mind. In S. Baron-Cohen, H. Tager-Flusberg, & D. J. Cohen (eds.), *Understanding Other Minds: Perspectives from Autism*. Oxford: Oxford University Press.

Mulholland, P. (1995a). *A Framework for Describing and Evaluating Software Visualization Systems: A Case-Study in Prolog*. Unpublished PhD thesis, Knowledge Media Institute, Open University, Milton Keynes.

Mulholland, P. (1995b). *Prolog Without Tears: An Evaluation of the Effectiveness of a Non Byrd Box Model for Students*. Paper presented at the Psychology of Programming Interest Group, University of Edinburgh, Edinburgh.

Nass, C., Steuer, J., & Tauber, E. R. (1994). *Computers are Social Actors*. Paper presented at the CHI'94.

Pain, H., & Bundy, A. (1987). What Stories Should We Tell Novice Prolog Programmers? In R. Hawley (ed.), *Artificial Intelligence Programming Environments* (pp. 119-130).

Papert, S. (1993). *Mindstorms: Children, Computers, and Powerful Ideas*. (Second edition). New York: Basic Books.

Perner, J. (1991). *Understanding the Representational Mind*. Cambridge, Massachusetts: MIT Press.

Resnick, M. (1994). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. Cambridge, Massachusetts: MIT Press.

Turkle, S. (1984). *The Second Self*. Simon and Schuster.

Wellman, H. M. (1990). *The Child's Theory of Mind*. Cambridge, Massachusetts: MIT Press.

View publication stats