Journal of Applied Logic 6 (2008) 526–533

Contents lists available at ScienceDirect

Journal of Applied Logic

www.elsevier.com/locate/jal

Metaphor in computer science

T.R. Colburn *, G.M. Shute [1]

Department of Computer Science, University of Minnesota, Duluth, MN, USA

ARTICLE INFO

Article history:
Available online 25 September 2008

Keywords:
Metaphor
Abstraction
Computer science
Ontology
Virtual entities

ABSTRACT

The language of computer science is laced with metaphor. We argue that computer science metaphors provide a conceptual framework in which to situate constantly emerging new ontologies in computational environments. But how computer science metaphors work does not fit neatly into prevailing general theories of metaphor. We borrow from these general theories while also considering the unique role of computer science metaphors in learning, design, and scientific analysis. We find that computer science metaphors trade on both preexisting and emerging similarities between computational and traditional domains, but owing to computer science's peculiar status as a discipline that creates its own subject matter, the role of similarity in metaphorical attribution is multifaceted.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

The language of computer science in general, and software development in particular, is laced with metaphor. Indurkhya [5] characterizes metaphor as "a description of an object or event, real or imagined, using concepts that cannot be applied to the object or event in a conventional way" (p. 18). In web applications, for example, it is common to refer to certain complicated data structures as *shopping carts*, even though within the application, a complex compendium of web pages, programs, and databases, there is nothing to which the concept of a shopping cart could conventionally apply.

There are many other examples. Computer users have incorporated *folders*, *directories*, *files*, *registries*, and *pages* into their language. Operating systems run *servers*, *clients*, and *daemons*. Application programs execute *threads*, occasionally experience memory *leaks*, and undergo *garbage collection*. Program exceptions are *thrown* and *caught*. Programmers organizing data regularly speak of *stacks*, *queues*, *trees*, *pipes*, and *streams*.

Algorithm designers manipulating data structures employ acyclic relations such as *parent*, *child*, *ancestor*, and *descendant*. Networks and systems of networks experience *congestion* and *flooding* and often require *filtering* or *flushing*. Networked applications *listen* on *ports* for *connections* with other programs by following a *protocol* beginning with a *handshake*. And finally, users of modern programs use pieces of screen displays referred to as *windows*, *buttons*, *tabs*, and *menus*.

We have described elsewhere [4] the role that *abstraction* plays in computer science. Our concern there was to characterize abstraction in computer science as distinct from abstraction in mathematics, and to indicate the key role that *information hiding* plays for abstraction in managing the complexity of programming languages, operating systems, virtual machines, and software architecture.

Many of the computer science metaphors enumerated above name abstractions in our sense. Here are three examples.

* Corresponding author.
E-mail addresses: tcolburn@d.umn.edu (T.R. Colburn), gshute@d.umn.edu (G.M. Shute).

[1] We would like to thank the organizers of the 2007 European Conference on Computing and Philosophy at University of Twente, Netherlands, and also Andrew Barsden, who made valuable comments on an early draft of this paper.

1570-8683/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.jal.2008.09.005

(1) A *shopping cart* in a modern web application is another name for an object that encapsulates data and operations unique to web commerce and that hides the details of its implementation from the many other objects it comes in contact with during the life of the application.
(2) A *thread* in a program is an object that a programmer can create to implement an arbitrary computational process. The programmer does not have to know anything about how to keep this program from interfering with the myriad other threads in the system, because a thread is an abstraction on top of the operating system, hiding the details of how the operating system schedules processes.
(3) Programmers needing to effect an exit from a procedure that involves a complicated transfer of processor control can *throw* an exception while needing to know nothing about how such an action affects the runtime stack.

In all of these examples, an abstraction is created to manage complexity through information hiding, and a metaphor is used to name the abstraction. It is important to note that the information hiding in each case could be achieved through the use of nonmetaphorical terminology to describe it. There is nothing germane to information hiding that requires metaphor, but using metaphor facilitates talk about it. One may also argue that many of the examples listed above, shopping carts among them, are not so much metaphorical attributions as they are descriptions of *virtual* versions of real things. This leads to several questions that are the subjects of this paper. What is the relation between computer science metaphor, ontology, and virtual entities? Do prevailing theories of metaphor capture the essential roles played by computer science metaphors? What role does abstraction play in computer science metaphors?

The concept of metaphor has not escaped philosophical scrutiny by philosophers, psychologists, linguists, and educators. So we approach these questions using the frameworks they have already erected.

2. Metaphor, language, and reality

While philosophers have considered the concept of metaphor since at least the time of Aristotle, modern treatments of metaphor can be situated in the general issue of the relationship of language and reality, as described by Ortony [11]. This is not to assume that these treatments drive a wedge between language and reality, as though they were distinct. One can argue that modern progress in philosophy of language has uncovered a consensus that language is itself part of reality. This should not stop us, however, from considering how language contributes to our knowledge of reality through metaphor. Indeed, as we see below, some theories of metaphor insist that metaphors are essential to knowledge.

In the extreme "nonconstructivist" view, language and reality are tightly related, with reality being literally describable by language properly used. Under this view metaphorical use of language characterizes rhetoric, not science. This view has its origins in the logical positivism of the previous century, according to which linguistic meaning is rooted directly in experience or logic; any other use of language is meaningless. This view has the consequence that any knowledge we gain through metaphor could have been gained without it.

In the extreme "constructivist" view, reality is only accessible through the interaction of basic given knowledge (sense perception) with preexisting and contextual knowledge; language actually requires metaphor in the creative construction of new knowledge. Under this view the distinction between literal and metaphorical meaning is blurred, as is the distinction between the language of the poet and the language of the scientist.

Theories attempting to answer the theoretical question "What are metaphors?" fall on the continuum between these extreme views [11, p. 11]. At one end (the extreme nonconstructivist end), metaphors are a purely linguistic phenomenon. Cohen [3], for example, proposes that metaphorical meaning can arise directly from literal meaning through a set of linguistic rules. In the middle of the continuum, metaphors are a general communication phenomenon associated with language use. For example, Searle [14] relates metaphors to the theory of indirect speech acts. At the opposite end of the continuum (the extreme constructivist end), metaphors are involved in the general phenomenon of thought and mental representation (Lakoff [7]), including scientific theory transmission (Boyd [2], Kuhn [6]). Metaphors such as the orbital model of the atom or string theory as a model of fundamental physics perform functions that cannot be fulfilled by literal language.

Computer science metaphors, when used to support abstraction as in the three examples above, seem to be paradigm examples of the constructivist approach to the relationship of language and our knowledge of reality. They expand the ontological framework of our language for talking about computational processes. They work their way into programming languages, the vocabulary of algorithms, and even the language of popular culture. But how do they work, and what exactly is their role?

3. Computer science metaphors and established metaphor theory

We will follow Indurkhya in calling the object or event being described through metaphor as the *target* and the concept(s) being unconventionally applied in the metaphor the *source*. As Indurkhya points out, general theories about how metaphors work divide into two groups, *comparative* theories and *interaction* theories. We will not attempt to characterize computer science metaphors as in general falling into one or the other of these groups. As we show below in Section 4, the roles of computer science metaphors are quite diverse, and both types of theories are applicable. To understand these roles, it is instructive to first understand these prevailing theories.

Comparative theories assume that metaphor arises from preexisting similarity between the source and target. Some computer science metaphors seem to exploit preexisting similarity, and some do not. For example, buttons of the type clicked by mouse pointers on graphical user interfaces, like a button to be pushed to start a computer pinball game, are very similar to real physical buttons, all the way down to the shading used to indicate whether the button has been pressed. However, other computer science metaphors do not exploit preexisting *physical* similarity. For example, when we describe a program as having a memory leak, there is nothing in the programmer's world that is "similar", in a physical sense of the word, to a malfunctioning faucet, or hose, or gasoline tank. The same is true for a programmer's idea of a thread and a host of others. It may be argued, however, that the programmer's world describes *abstractions*, for example, a memory store, that *are* similar to physical source concepts, for example, a storage tank. Whether these similarities are preexisting, however, depends upon the type of role the metaphor is playing, as we describe later in Section 4.

Interaction theories of metaphor claim that metaphors work through interaction between the source and target, and not through preexisting similarity. Indurkhya characterizes interaction theories as follows: "[I]nteraction theory proposes that every metaphor involves an interaction between its source and its target, a process in which the target (and possibly the source) is reorganized, and new similarities between the source and the target emerge" (p. 3).

Black [1] describes this interaction as "projecting upon" (itself a metaphor) the target a set of "associated implications" that are predicable of the source. He uses "Marriage is a zero-sum game" as an example. It uses source concepts such as "contest", "opponents", and "winning" and projects them onto marriage as, for example, "A marriage is a sustained struggle between two contestants in which the rewards (power? money? satisfaction?) are gained only at the other's expense" (p. 28).

Indurkhya is concerned that the nature of this projection relation in interaction theory is vague and unconstrained. If metaphors are unconstrained "then anything would be meaningful and there would be no way to maintain a distinction between what is genuinely metaphorical and what is certainly nonsense" (pp. 3–4). Beyond this problem, traditional interaction theory applied to some computer science metaphors does not seem to work, because "new similarities between the source and target" do not emerge through the use of the metaphor; rather, the metaphor is introduced "after the fact". Independent computational processes competing for operating system resources were around for a long time before they were called "threads". Addresses of memory locations were involved in computer programs long before they became "pointers". What induced the introduction of these metaphors, and others like them, if not preexisting or emerging similarity?

Interaction theories of metaphor assume that the source and target concepts already exist, and that the question is how new similarities between them come into being. Indurkhya quotes a Carl Sandburg poem in which "The fog comes on little cat feet…". The concepts *fog* and *cat* are well established, but the similarity between them is created by the poem's imagery of silent creeping.

The reason that interaction theories do not seem adequate for some computer science metaphors is that the assumption that source and target concepts already exist is incorrect. Computer science is a discipline that *creates its own subject matter*. Computers and computational processes are both studied *and* created by computer scientists. Programming languages and data structures are created and then used to create and study others. Natural scientists build and use tools to study nature, not the tools themselves. Computer scientists create computers and programming languages, study them, and, through a bootstrapping process of continuous abstraction, create more. So the role of metaphor in computer science is intimately related to computer science ontology.

Ontology, in its traditional, philosophical meaning, is the study of what there is. As such it is often viewed as a metascience, where "what there is" encompasses not just nature but also cognition, mathematics, aesthetics, etc. As observers of what there is, philosophers are often careful to assume that they do not create what they study. What there is, in any realm, is out there to be discovered.

Indurkhya, for example, when analyzing cognitive relations between a subject's concept network and the environment with which the subject interacts, is careful not to claim that reality itself is constructed through a subject's interaction with the environment. What is constructed is the subject's ontology through changes in the correspondence of his concept network to the environment. Indurkhya finds metaphor essential to cognition since new concepts are learned through metaphorical projection. "[P]rojection works by indirectly altering the structure of the environment so that it better fits the structure of the concept network. I say 'indirectly' because what the cognitive agent is actually doing is giving a new ontology to the environment, and it is reality that is fitting this ontology with a structure" (p. 165).

This account makes sense when the agenda is primarily epistemological, that is, when we are trying to account for new knowledge of existing reality (new ways of looking at cats and fog). But the subject matter of computer science is a product of its own creation. Transistors, computers, and programming languages do not exist prior to their conceptualization. Such conceptualization does not consist in new knowledge of old reality, but in new reality seeking a conceptual framework in which to be situated. Computer science metaphors provide such a framework by exploiting both preexisting and emerging similarity, and also by enforcing similarity in the worlds it creates.

In the next section we will show how some computer science metaphors are used in a *pedagogical* role to explain the increasingly complex components—the machines, languages, and data structures—created in computer science. We will also show how other metaphors play an important role in the *design* of such components. Furthermore, as computer scientists develop more complex designs, they must develop scientific theories governing their components. Here again, metaphors play a role, a *scientific* role that explains how we might improve our designs to make them faster and more efficient.

4. The roles of computer science metaphors

4.1. The pedagogical role

Metaphors serve a pedagogical role for both users and creators of software. For users, consider an example. When a web shopper encounters a button, menu item or icon labeled "Add to Shopping Cart" it suggests that shopping from that web site will work just like shopping in a grocery store. Using knowledge of ordinary shopping, the web shopper can infer that the overall interaction will be broken into two phases: selecting the items to be purchased (putting them into a shopping cart), and paying for the items (checking out). The web shopper expects that somewhere readily accessible, there is also a button, menu item, or icon for checking out.

This effect is a direct result of the metaphorical description of the target concept, namely online shopping, in terms of the source concept, namely ordinary shopping. As Indurkhya remarks, "The source does not just include the concepts mentioned in the description, *but other related concepts as well*" (p. 18, emphasis added). So although the four-word target description "Add to Shopping Cart" does not mention a checkout process, the web shopper expects one. With just these four words in the user interface, the shopper gets as much information as a couple of paragraphs of textual documentation or tutorial training. This is due to an intentionally designed similarity between shopping in a physical store and shopping in an online store. For the user of the online store, this similarity is preexisting. For the designer of the website, this similarity is enforced in the website's programming.

In other cases, metaphorical similarity emerges over time and is only visible to programmers, as in the following example. Prior to the introduction of modern programming languages, programmers would sometimes have to write code that performs what are known as *dynamic non-local exits* of an executing procedure. Such a procedural exit is brought about by an exceptional event (such as a system error or user abort) that requires non-normal processing. One important characteristic of exceptions is that they are often discovered at lower levels of complicated code, but cannot be handled there in a manner that is meaningful to the programmer. The meaning is usually only present in higher level code, which calls upon the low level code through a sequence of subprogram calls.

Bringing about such an exit without support from a programming language requires the programmer to know a precise location on a runtime structure known as the *control stack* (itself a metaphor), and the code produced needs to ignore a number of pending actions represented on this stack and give control to the precise location in question. Understanding the action required, let alone writing code to do it, is difficult.

As a result, modern programming languages introduce the metaphor of *catch and throw* to describe this situation. The metaphor both (i) abstracts the action required into high-level programming languages, and (ii) explains what is happening in the low-level language so programmers can understand the consequences of the code they write. "Explains" is not used here in the sense of providing causal conditions, but rather in the sense of merely aiding the understanding, of "shedding light"—something most software developers desperately need while learning to program, and when learning a new programming language.

Users of new software may be like children when it comes to using the tools in the software's interface. Programmers who are new to programming or who are just learning a new language may also be like children when it comes to applying the language's concepts. The acquisition of conceptual frameworks by children has been described by Piaget [12,13], and Indurkhya builds on Piaget in constructing a general interaction theory of cognition. Indurkhya's interest in Piaget stems from the latter's description of an organism's adaptation to its environment, specifically, cognitive *assimilation* and *accommodation* in children. Assimilation is "the process by which a cognitive agent sees every situation, every environment, through the structure of its preexisting concepts, or *schemas*" (p. 120). Indurkhya borrow's Piaget's concept of assimilation but calls it *projection* in the sense of Black. Examples from the study of children include situations of pretending, as when a child plays with a dollhouse by projecting her schema of ordinary life in a home onto her dolls and miniature furniture. When a user of web site software first encounters a web shopping cart, she must project her schemas involving ordinary shopping carts onto the web shopping cart. Similarly, when a programmer first comes in contact with the *catch/throw* concept in exception handling, she must project her ordinary schema of catching and throwing discrete physical objects to her technical understanding of a runtime processing stack. Such projection results in the emergence of a new kind of similarity between these disparate domains.

But assimilation/projection is only half the story of children's cognitive adaptation. "[A]ssimilation alone produces only a playful behavior and not an 'intelligent' one. The unexpectedness of response is lost unless it is *integrated* into the conceptual organization by suitably modifying it. Precisely this task is accomplished by *accommodation*. … It is the accommodation that gives the cognitive agent a capacity to 'learn' "(p. 120). For example, even a very young child has a well-developed 'sucking schema' according to which he will grasp an object and bring it to his mouth. In time, however, as the child's visual apparatus becomes more acute, this schema may give rise to a new one that causes responses that bring objects into the child's visual field and not his mouth. His conceptual network (Piaget's *schemata*) has accommodated to his environment.

The quality of a pedagogical metaphor depends on how much accommodation is required. For example, using an online shopping cart metaphor requires some accommodation. A user must know to click on an icon using pointing devices (mice, trackballs, and touchpads). *Pointing* in the context of a computer user interface is itself a metaphor whose value lies in giving the user the ability to, for example, select items from a list or move objects around on the screen. The metaphor requires some accommodation: clicking a button replaces grasping an object and visual feedback comes from an arrow (mouse

cursor) painted on the screen instead of from a finger. But the success of the metaphor indicates that these accommodations are relatively easy for most people.

Similarly, when a programmer has learned the *catch/throw* language construct by projecting her source concept of physical catching and throwing onto her concept of a runtime stack, she will come to readily use the construct in various diverse program contexts without trading on the metaphor at all. When this occurs, the issue of whether the metaphor is about preexisting or emerging similarity is no longer relevant. The programmer's conceptual network will accommodate the *catch/throw* construct as simply a technical abstraction for a complex transfer of control. Her understanding and utility with the concept will not depend on any metaphorical projection. Facility with the new concept unencumbered by metaphorical baggage is part of what it means to be a programming expert.

The ideal metaphor is one that requires no accommodation at all. Cache memory is an example. The *cache* metaphor trades on the source concept of specialized storage. However, a programmer accesses memory through a cache transparently, meaning that the interface through a cache is exactly the same as the interface without a cache. The only difference is in the speed of the response, which is faster with a cache.

Virtual memory is another example. Here again, a programmer accesses memory addresses using exactly the same interface as without virtual memory. However, the memory addresses are virtualized: they are translated into physical addresses. The point of doing this is that the translation can be done differently for different processes in the computer. The operating system manages the translation so that the same address in two different processes are translated into different physical addresses. This allows programmers (or compilers) to allocate memory without concern for other processes that are using the same physical memory.

This kind of virtualization is common in operating systems. One common feature of operating system virtualization is creating ways of using resources such as memory, processor time, and space on a display, that allow programmers to write software as if their code would be the only code running on the machine. They do not have to make accommodations for other programs that are running at the same time.

4.2. The design-oriented role

The pedagogical role for computer science metaphors can be regarded as an explanatory role, for they help explain concepts or phenomena to software application users or application programmers. But computer science metaphors also serve an important but distinct purpose for those engaged in the creative activity of hardware production and software tool development. These are the creators of devices, languages, algorithms, or data structures who need a terminology or context in which to embed them. They create the metaphors that others use. Both hardware and software designers, for example, borrowed the concept of a *port* to describe certain kinds of system interfaces. Programming language designers came up with *thread* and *catch/throw*. Graphical user interface designers came up with *button*, *menu*, and *shopping cart*. As we next describe, these metaphors can be seen as design objectives.

A software developer today is very often engaged in designing an interface for an intended user. A useful approach is to first create a metaphor whose target domain (the user interface) is close to a source domain that is familiar to the intended user, and then to implement the interface in software. In such a case, again, an appeal is made to preexisting similarity. For example, the *desktop recycling bin* metaphor functions in this way for a programmer. For many such "design metaphors", the target domain is suggested by looking at the world from the point of view of a user. A developer of a design metaphor approaches metaphors in a "backward" direction, looking first at a target domain (e.g. a desktop user interface), then locating a source domain for it (e.g. a recycling bin in a kitchen closet).

Design metaphors are employed not only in the creation of user interfaces but also in programming language development. For example, early programming languages were plagued by problems stemming from the fact that different parts of a large program often want to use the same name for separate program locations. Ensuing languages solved this problem by creating a level of abstraction that separated large programs into distinct "name spaces" and calling them *packages*. This metaphor served not only to allow the same name to be used in different name spaces, but also to erect a barrier (a package wrapper) around a space to eliminate confusion.

While we used the *catch/throw* metaphor above to describe a pedagogical role for metaphors, it is also an excellent example of a design metaphor (computer scientists are not afraid to mix their metaphors!). As mentioned above, programmers need to deal with exceptions in order to write robust code. In the beginning, such mechanisms made use of a *jump* metaphor that is implicit in the understanding of all machine code. In order to capture the concept of transfer of control from one part of a program to another, nonadjacent part, primitive programming languages implement instructions whose mnemonic devices involve the concept of jumping.

One mechanism for dealing with exceptions, in particular, used a *setjmp* call to record the state of execution in higher level code in a structure that can be accessed by lower level code. If an exception is detected in lower level code, that code can make a *longjmp* call with that structure to resume execution from the location of the *setjmp* call. This mechanism is fraught with difficulties. For one thing, there are usually several different types of exception to be handled and there is a need for making all of them available to lower level code. A more serious problem is that the low level code must make the choice of which *longjmp* call to use among possibly many. This creates an unwanted dependence between the low level code and the high level code, increasing its complexity while decreasing its maintainability and modifiability.

As often happens when people deal with problems, the first solutions are inspired by existing mechanisms. Programmers, from the days when assembly language was the first language they learned, based the *setjmp*/*longjmp* mechanism on the assembly language concept of a *jump* instruction. The characteristic of the jumping metaphor that caused the problem is that a *jump* instruction always specified a definite target. When one jumps, one jumps from a specific *here* to a specific *there*. The idea of jumping to an unknown place did not fit into that conceptual framework.

In the case of the *catch/throw* metaphor, the metaphor treats throwing and catching as independent actions: you can throw something without regard for who will catch it. (Consider the "alley-oop" play in basketball, where one player simply throws the ball toward the basketball hoop and lets another player find his way to the hoop at the same time that the ball arrives in order to execute a slam-dunk.) This is the key to eliminating the unwanted dependence and resulting programming complexity. Use of the *catch/throw* metaphor aided the design of programming languages to replace languages using *setjmp* and *longjmp* constructs. With this replacement emerged new similarities between the programmer's abstract world of control flow and the world of physical activity. This example shows that, again, both comparative and interaction theories of metaphor are relevant to computer science.

Design metaphors that use new target domains in this way are, in effect, paradigm switches. They require a new approach to a problem, and consequently take longer to develop. They are more general than design metaphors that arise directly from the requirements of a particular software project. The history of operating systems has many examples of such metaphors: files and folders for system users, and windows, event queues, and many others that are of interest only to programmers.

4.3. The scientific role

While the *catch/throw* metaphor can function as a pedagogical tool and a design objective, other metaphors aid in scientific explanation. Computer science, like any science, can involve the search for explanations of observed phenomena. For example, the study of both individual machine organization and networks of many machines raises questions about performance and resource utilization. What is the maximum throughput of a system? Why are delays experienced at certain nodes of a network? To answer these kinds of questions, a *flow* metaphor is employed. Here, the target domain—it could be information flowing through a network or instructions flowing through a processor—is mapped to a more familiar source domain such as water flowing through a pipe. In fact, this metaphor is so commonplace that other source domains could be used, for example, the flow of electrons through a wire. In any case, the metaphor serves to explain the target domain in terms of the source domain, and the explanation depends on preexisting similarity between the domains.

The flow metaphor has also been enriched with a variety of accommodations. Continuous flows of one, two, or three dimensions have been studied extensively in the mechanics of continua, an area of physics. Discrete flows, such as flows of packets in a network or flows of instructions through a pipelined processor, are analysed in the mathematical area of queuing theory. Although the concept of a *queue* of information may be regarded as relying on metaphorical projection, in mathematical practice it has transformed into a formal analytic tool for scientists.

Much of the success of flow language in other disciplines is due to the existence of natural conservation laws that allow natural scientists to formulate precise mathematical conditions on the flows and to design tools and devices that depend on these laws. But in a science that creates its own subject matter, the design and the science are inseparable parts of the same thing. However, metaphors can help computer scientists treat their subject matter scientifically. In computer networks, for example, there is no natural law of conservation of information. However, network programmers would like to have one, so they make things work that way, designing error detection and correction algorithms to ensure that bits are not lost during transmission. Their conservation "law" becomes a principle of design. In this case, a new similarity between the source and target domain does not so much as *emerge*, as become *prescribed* by a design, and then *enforced* by programming.

The flow metaphor also finds its way into programming areas that do not have to do with networks or processor architecture. As mentioned previously, good programmers are very concerned about memory "leaks", a condition that will cause a program to eventually run out of memory the way a leaky gas tank runs out of fuel, causing the program to grind to a halt. To avoid this, programmers design their code according to a "law of conservation of memory", requiring them to fastidiously return memory that is no longer needed by their programs back to the system. Again, a new kind of similarity becomes prescribed and enforced.

Programmers are taught to identify individual conservation "laws" that must be enforced before they design procedures that describe computational processes. These laws are called *invariants* and are used to ensure that the procedure correctly carries out a process. As such, they do not describe the way things *are* like natural laws do. Instead, they prescribe the way things *should be*. For example, when summing up a list of numbers, an invariant might state that a variable contain a correct partial sum at each step, and that any references to elements on the list are within acceptable bounds.

The *out of bounds* metaphor is an essential tool for stating invariants that are critical to ensuring that programs are not vulnerable to being compromised by "hackers" (in the pejorative sense of the term). Programs that ignore this metaphor are susceptible to hacker attacks that exploit conditions that are themselves characterized metaphorically as *buffer overflows*. Such attacks can be high-profile and have significant economic costs. One of the objectives of software engineering is to identify and enforce invariants that imply both that a given procedure meets the requirements for which it is written and that data security and integrity are maintained. Toward this end, metaphors help programmers treat their discipline

scientifically, so to benefit from the attendant formal and analytical tools and methods, by prescribing similarities that must be enforced in the programmers' computational worlds.

5. Metaphorical attribution and virtuality

Computer science is awash with not only metaphors, but also virtual entities. (The use of "virtual" here is distinct from the more technical description given above whereby an operating system employs virtualization techniques to give multiple processes the impression that they each have exclusive access to computational resources.) A classic example is the online shopping cart, but the variety of virtual entities is limitless, ranging from the simplest of user interface items like buttons and menus, to containers like files and folders, to quasi-geographic locations like chat rooms and shopping malls, to virtual persons in the form of avatars. In each of these examples, understanding the usefulness of the virtual entity trades on metaphorical projection from an ordinary, well-known realm to a cyber realm. However, it can be argued that to use an online shopping cart, for example, is not to use a metaphor at all, but to use a virtual version of a real thing. What is the role of metaphorical attribution in the creation of virtual entities?

While Indurkhya is interested in the ways interesting similarities are *created* through metaphor, as in the Sandburg poem where the fog and a cat are likened through their silent creeping, the virtual entities of computer science use preexisting similarity to feature a cyber world. What is created through the metaphor is not similarity but virtual ontology. The metaphorical role of the icon labeled "Add to Shopping Cart" is to use similarity to allow inferences in a virtual environment.

Moor [9] has considered the implications of underestimating the changes to our conceptual framework brought about by computers. These changes are due in part to what Moor calls the *logical malleability* of computers, or their ability, despite their fixed physical makeup, to perform widely diverse functions. This diversity of functionality accounts for the diversity of virtual entities that computers can encompass.

The effort to give new functionality to a computer can arise from a desire to, in Moor's terms, "informationally enrich", or increase the role of information in, a traditional area of endeavor. Moor's examples include sweeping institutional changes from the gold standard to a cashless society, from traditional to information warfare, and from physical to online privacy. But informational enrichment also marks the change from our use of concrete objects like folders, shopping carts and malls to our use of virtual versions of them. The more informationally enriched a traditional concept is, the more virtual it becomes, and the less metaphorical its use is.

When a traditional concept begins the process of becoming informationally enriched, metaphor is useful, though not essential, in tying it to a source concept with which everyone is familiar. However, virtual entities are not burdened by the physical constraints of their noninformational counterparts, and metaphors tying them to physical sources may even get in the way. For example, the now familiar concept of a computer *file system folder* began as an informationally enriched version of a physical file cabinet. But the concept of a virtual folder has grown beyond its original constraints, as users now take for granted the ability to create an unlimited nesting of folders within folders, which would quickly prove to be impractical with files and folders in a file cabinet. When informational enrichment of a traditional source concept has gone this far, the associated metaphor may have started its life based on preexisting similarity to the target concept, become more developed through emerging similarities, and wound up, as in the case of the unlimited nesting of folders, contributing to a measure of *dis*similarity between the source and target.

Traditionally, a metaphor becomes "dead" when the application of the source concept to a target domain is no longer unconventional. But virtual entities suggest another way: when the target domain becomes so dissimilar to the source through informational enrichment that a metaphorical name for the target concept ceases to be metaphorical and becomes an historical artifact. It may be the fate of many names of virtual entities to become dead metaphors in this sense.

6. Metaphor and abstraction

We have argued elsewhere [4] that the primary subject matter of computer science is *interaction patterns*. This is in distinction with mathematics, whose primary subject matter is *inference structures*. Thus understood, the difference between computer science and mathematics can also be characterized through their abstraction objectives. Since the interaction patterns studied and produced by computer science are of ever-increasing complexity, the abstraction objective of computer science is *information hiding*. Since the inference structures studied and produced by mathematics depend on formal properties and eschew empirical content, the abstraction objective of mathematics is *information neglect*.

Fundamental differences between computer science and mathematics can be further highlighted through their use of metaphor. Lakoff and Núñez [8] address cognitive and ontological questions about mathematics through the notion of *conceptual metaphor*, "a cognitive mechanism for allowing us to reason about one kind of thing as if it were another. … '[C]onceptual metaphor' [is a] neural mechanism that allows us to use the inferential structure of one domain (say, geometry) to reason about another (say, arithmetic)" (p. 6). They give examples such as the metaphor of a measuring stick to grasp the arithmetic of fractional and decimal numbers. These examples support their claim that the primary function of conceptual metaphors is "to allow us to reason about relatively abstract domains using the inferential structure of relatively concrete domains" (p. 42).

Since the abstraction objective of mathematics is information neglect, mappings from measuring sticks to fractions can occur. If you abstract the essence of a measuring stick, the result is a physically featureless magnitude to which one can apply the divisions necessary to create fractions.

But since the abstraction objective of computer science is information hiding, the role of metaphors is the opposite of that described by Lakoff and Núñez. Their primary function is to allow us to reason about relatively concrete and very complex domains (like manipulating a runtime control stack) using the inferential structure of relatively simple and abstract domains (like catching and throwing).

Another example is provided by the programmer's concept of a *thread*. The source domain of this metaphor is interesting in that it is itself a metaphor: a "thread" of conversation or a "thread" of thought. These ideas go beyond the mere notion of sewing thread and include the implications that it (i) is unbroken, (ii) leads from one thing to another in sequence, (iii) interacts with other things that are not part of it during its sequence, (iv) draws things together, and (v) is traceable in principle. The fact that these features also apply to a programmer's notion of a thread suggests that there is a metaphor schema (to use Piaget's terms) at work; a schema for producing metaphors with some commonality. That commonality is a developing abstraction.

To see this abstraction for the programmer's thread metaphor we need to look at the surrounding context. Part of the thread metaphor is the notion of linearity: leading from one thing to another in sequence. But a thread is also often a part of a more complex order structure: a fabric. There may be a more or less conscious heuristic principle implicit in the thread metaphor for the analysis of complex orderings (fabrics): decompose them into more or less linear orderings, i.e. threads. This decomposition takes an outwardly chaotic and difficult to manage space of competing and cooperating computational processes and hides its complexity behind an abstraction that is expressed as a metaphor.

7. Conclusion

We have seen how computer science metaphors provide a conceptual framework in which to situate constantly emerging new ontologies of computational environments. We observed this through the specialized roles that computer science metaphors play for software users and creators in learning new concepts, for language and machine developers in designing new tools, and for computer scientists in creating their own conservation "laws".

The richness and diversity of computer science metaphors reveals that both comparative and interaction theories of metaphor are at play in their deployment, so that both preexisting and emerging similarities between source and target concepts are involved. But owing to computer science's peculiar status as a discipline that creates its own subject matter, metaphorical similarity in computer science is a complex affair. We saw that technical expertise in programming can make pedagogical similarity irrelevant, that programming as a science can produce metaphors that function as laws to be enforced, and that informational enrichment of a source concept can produce a virtual entity whose initial similarity to the source actually evolves into elements of dissimilarity.

We have also seen that the use of metaphor can help attain the abstraction objective in computer science in a way that further distinguishes computer science from mathematics. Progress in computer science is facilitated by the hiding of complexity through increasingly higher level abstractions, and many such abstractions begin their lives as metaphors.

References

[1] M. Black, More about metaphor, in: [10], pp. 19–41.
[2] R. Boyd, Metaphor and theory change: What is "metaphor" a metaphor for?, in: [10], pp. 481–532.
[3] L.J. Cohen, The semantics of metaphor, in: [10], pp. 58–70.
[4] T. Colburn, G. Shute, Abstraction in computer science, Minds and Machines: Journal for Artificial Intelligence, Philosophy, and Cognitive Science 17 (2) (July 2007) 169–184.
[5] B. Indurkhya, Metaphor and cognition, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.
[6] T. Kuhn, Metaphor in science, in: [10], pp. 533–542.
[7] G. Lakoff, The contemporary theory of metaphor, in: [10], pp. 202–251.
[8] G. Lakoff, R.E. Núñez, Where Mathematics Comes From, Basic Books, New York, 2000.
[9] J. Moor, Reason, relativity, and responsibility in computer ethics, in: [15], pp. 40–54.
[10] A. Ortony (Ed.), Metaphor and Thought, Cambridge University Press, Cambridge, 1993.
[11] A. Ortony, Metaphor, language, and thought, in: [10], pp. 1–16.
[12] J. Piaget, The Origins of Intelligence in Children, W.W. Norton, New York, 1963.
[13] J. Piaget, The Psychology of the Child, Basic Books, New York, 2000.
[14] J.R. Searle, Metaphor, in: [10], pp. 83–111.
[15] R.A. Spinello, H.T. Tavani (Eds.), Readings in Cyberethics, second ed., Jones and Bartlett, Sudbury, MA, 2004.