Virtual machines and program comprehension

Josh D. Tenenberg
Department of Mathematics and Computer Science
Indiana University at South Bend
1700 Mishawaka Avenue
South Bend, IN 46615
(219) 237-4525
tenenberg@acm.org

1 Overview

The objective of this paper is to critically examine several claims from the Psychology of Programming literature on the mental representations and processes involved in computer program understanding, and to suggest a reinterpretation of these claims in light of a set of core constructs from computer science. The claims concern whether mental simulation is a viable inference strategy as programs grow in size, that domain knowledge is crucial to the understanding of a program, that program readers develop a two-level representation of the program, and that program readers use a predominantly bottom-up or top-down inference strategies. The set of core computer science constructs involve the view of programs as a set of hierarchically arranged virtual machines, each machine having its own description. My reinterpretation centers around the claim that program readers are able to understand a virtual machine at a single level of description.

I will present my argument by first providing two short example programs, inviting the reader to make a number of inferences about these programs, foreshadowing the claims that I subsequently propose. I will then define what I mean by virtual machine in terms of the reduction of complexity through abstraction and naming, and argue through reference to computer science literature that this is a central, if not *the* central concept in computer science. I will then return to each of the psychological claims in turn, citing the empirical research supporting these claims, and offering a reinterpretation in light of the hierarchical virtual machine view.

2 Two example programs

Consider the code in Figure 1 and Figure 2. The first is the main routine for a Tic-Tac-Toe program, while the second performs a depth-first search of a graph.

```
main()
{

  board_type Board;
  player_type Player;
  move_type Move;
  player_type Winner;

  do // once per game
    {
      Initialize ( Board );
      Print ( Board );
      Initialize ( Player );
      Winner = NO_WINNER;

      do // once per move
        {
          Get_legal_move ( Board, Player, Move );
          Make_move ( Board, Player, Move );
          Print ( Board );
          if ( test_for_win ( Board, Player ) )
            Winner = Player;
          else
            Player = next_player (Player);
        } while ((Winner == NO_WINNER) && !test_for_draw(Board));

      if (Winner == NO_WINNER)
        Draw_message();
      else
        Win_message(Winner);

    } while ( want_to_play_again() );
}
```

Figure 1: Tic-Tac-Toe Program

```
void dfs(int visit_num[], graph & G, int Mark[], vertex current_vertex)
{
  static count = 0;      // Numbers each vertex in order visited
  vertex v,w;
  Stack S;               // Standard Stack operations are defined

  Mark[current_vertex] = true;
  S.Push(current_vertex);
  while (! S.Is_Empty())
    {
      v = S.Pop();
      count++;
      visit_num[v] = count;
      // Iterate through all nodes adjacent to v
      for (G.First(v); !G.End(v); G.Next(v))
        {
          w = G.Current_Edge(v);    // w is connected to v by an edge
          if (!Mark[w])
            {
              Mark[w] = true;
              S.Push(w);
            }
        }
    }
}

main()
{
  graph G;
  int num_nodes = G.Number_of_nodes();
  int * visit_num = new (int [num_nodes]);  // Order in which visited
  int * Mark = new (int [num_nodes]);
  vertex i;

  for (i = 0; i < num_nodes; i++)
    {
      visit_num[i] = 0;
      Mark[i] = false;
    }

  // Do a dfs on each connected component
  for (i = 0; i < num_nodes; i++)
    {
      if (!Mark[i])    // Node has not yet been visited
        {
          dfs(visit_num, G, Mark, i);
        }
    }
}
```


Figure 2: Depth-first Search of a Graph

Although neither is a complete program at the level of C++ language primitives, it is not difficult to develop a coherent understanding of the algorithms at the level at which they are presented.

For example, we can easily mentally simulate the operation of each algorithm. Note that this is regardless of any property, such as size or complexity, of the underlying implementation. That is, the simulation is performed based on beliefs about the meanings of the terms of the description language derived from previous knowledge about similarly named objects. But this also means that our mental representations of these programs are not constructed in a bottom-up fashion, since, in neither case have I provided the code implementing the bottom levels of these programs.

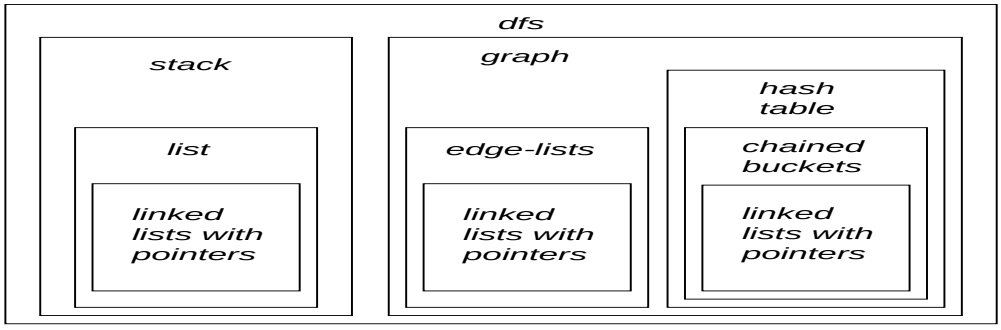

Figure 3: DFS virtual machine hierarchy

Figure 3 gives a graphical view of the virtual machine hierarchy for the complete depth-first program that I have implemented[1], down to the level of C++ language primitives and standard C and C++ libraries. My claim is that the mental representation that a programmer would possess upon reading all of

[1]The C++ code for the depth-first-search can be obtained by emailing the author, or from http://phoenix.iusb.edu/josh/code/dfs

```
int bucket_chain::Find( const char * key )
{
  for(Node* tmp = List_Head -> Next; // Dummy list header
      tmp != NULL; tmp = tmp -> Next )
    if(!strcmp(tmp -> Element, key))
      {
        Current_Pos = tmp;
        return true;
      }
  return false;
}
```

Figure 4: Find an element of a list

the code that this figure represents would not simply be two-leveled, one for the program and the other for the domain, but would be multi-leveled, reflecting the hierarchical structure of the implemented abstractions.

Not only is the highest level algorithm understandable without considering the lower level implementation, but conversely, the lower levels can be understood without consideration of the higher levels. This implies that knowledge of the domain of application of the program often has no bearing whatsoever on understanding lower levels of description. For example, the code fragment in Figure 4 implements the find function for the chained bucket. Understanding how this function operates neither requires nor is aided by knowledge of the graph program in which it is embedded.

3 Virtual machines

Dietrich [Die94, p13] defines virtual machines as follows.

> General-purpose computers are programmed ... [by being] given a description of a computation – of another machine – and the input for that other machine. Such a description is what a program is. The computer then runs the described machine on its input. This may not seem to be what's going on when you run your word processor, but it is. When a computer is running another described machine, the latter is called a *virtual machine* with respect to the base computer. Any time a software package executes, a virtual machine comes into existence.

Consider again the depth-first search algorithm as presented above in Figure 2. The defined data abstractions (i.e., stack and graph) are treated as primitives of the programming language at this level of description. The "virtual" aspect of this description is that there exists an interpreter for the primitives of this description language through one or more levels of software and hardware, that maps descriptions of machines to executions. Although I have chosen to

implement the stack and graph in software, nothing in principle prevents me from implementing this interpreter directly in hardware. Likewise, nothing in principle prevents me from building a "Pascal machine", or a "C++ machine"; it is for pragmatic reasons that I use layers of software to map my algorithms to executions.

4 Virtual machine languages and programming languages

The way that programming languages support the construction of virtual machines is in their ability to extend the language beyond what is provided by the primitives. To put it simply, a programming language is only as useful as the linguistic abstractions it enables a programmer to construct. Hal Abelson writes [FWH92, Forward] "Perhaps ... future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application." Every time that a programmer writes a new program, they consider how they will represent – and name – objects and operations that implement the desired functionality. Most high-level languages provide a facility for naming and referencing structured data types, consisting of other (primitive and constructed) named data types, a facility for naming and referencing new procedures, constructed from other program statements possibly including other named procedure references, and a macro-style capability. These recursively defined syntactic constructs give rise to structures and procedures of arbitrary complexity.

For example, we can take the following as a rough definition of a virtual machine description for a lexically-scoped language. A virtual machine description consists of a possibly empty set O of variables, named and typed, and a non-empty set F of functions, with return values and formal parameters also named and typed. Additionally, all variable references within the functions of F are either bound to the local variables and formal parameters, or globally bound to the variables in O. Relating this to the depth-first search example, the presented algorithm is itself a virtual machine description, as are the stack and graph. However, the `bucket_chain::Find` function is not by itself a virtual machine description, since `List_Head` and `Current_Pos` occur free (not locally bound) within it. However, this function is *part* of the description for the `bucket_chain` virtual machine.

Rather than thinking of programs as being constructed of virtual machines, they can be equivalently viewed as hierarchically arranged function and data abstractions. A more formal treatment of data abstractions can be found in [FWH92].

Programming languages can be distinguished in terms of the kinds of object that can be combined and named, as well as rules for determining the scope of names. A few examples should suggest how programming languages differ in their support for constructing new virtual machine languages. For example,

C++ enables function definitions to be named fields of named structured data types, whereas Pascal and C do not. C permits arrays to be returned from functions, whereas Pascal does not. Scheme permits functions to be returned from functions, whereas neither Pascal nor C does. More detailed descriptions of language-extensions and scope can be found in most programming language books, such as [FWH92].

To the extent that two programming languages permit the same kinds of description languages for a particular virtual machine, they will differ little to the programmer at the level of that description language. For instance, given that the requisite stack and graph data abstractions and procedures have been defined, the depth-first search program above would look much the same whether in C, Pascal, Modula, Mesa, Ada, or any of a number of block-structured imperative languages, save for trivial syntactic variations. Thus, languages that look different at the primitive level may look nearly identical at the highest levels of virtual machine description.

5 The virtual machine model in Computer Science

The virtual machine view that I have described is considered by many to be the very cornerstone of Computer Science. Guy Steele describes the importance of abstraction and language extension [SF89, ppxv-xvi].

> The most important concept in all of computer science is abstraction. Computer science deals with information and with complexity. We make complexity manageable by judiciously reducing it when and where possible. ... Abstraction consists in treating something complex as if it were simpler, throwing away detail. In the extreme case, one treats the complex quantity as atomic, unanalyzed, primitive. ... Naming is perhaps the most powerful abstraction notion we have, in any language, for it allows any complex to be reduced for linguistic purposes to a primitive atom.

Hal Abelson focuses on the language issue, particularly the virtual aspect of interpreters and the manner in which languages can be hierarchically defined [FWH92, Forward].

> the most fundamental idea in computer programming: [is] The interpreter for a computer language is just another program. ... But that program is written in some language, whose interpreter is itself just a program written in some language whose interpreter is itself ... Perhaps ... future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application.

Goodman and Miller focus on the hierarchical aspect of constructing abstractions, and relate this to reducing complexity (emphasis in original) [GM93, p4,5].

> Perhaps the most important contribution to come out of Computer Science to date is a better understanding of complexity. The use of abstractions is one of the keys to understanding. ... A series of abstract models has been developed to help with this problem [of complexity]. ... The models are hierarchical, that is, models are built upon lower-level models. ... a **procedure** or **function** allows the programmer to create an operation of arbitrary complexity. ... Of course, a procedure may invoke another procedure, which may in turn invoke yet another, and at each level the function is defined, but the implementation of the function is obscured.

Finally, Tanenbaum discusses the relationship between virtual machines and virtual machine languages [Tan90, pp3,4].

> There is an important relation between a language and a virtual machine. Each machine has some machine language, consisting of all the instructions that the machine can execute. In effect, a machine defines a language. Similarly, a language defines a machine – namely, the machine that can execute all programs written in the language. ... A computer with n levels can be regarded as n different virtual machines, each with a different machine language. ... A person whose job it is to write programs for the level n virtual machine need not be aware of the underlying interpreters and translators. The machine structure ensures that these programs will somehow be executed. It is of little interest whether they are carried out step by step by an interpreter which, in turn, is also carried out by another interpreter, or whether they are carried out directly by the electronics. The same result appears in both cases: the programs are executed.

The key issues for computer scientists have thus concerned the ability to construct new machines and languages, each subsequent machine described in terms of previously defined languages. For those studying psychology of programming, the key issue involves how people construct the meanings of the terms of each of the virtual machine languages.

6 Implications of the virtual machine model

All programs can be viewed as descriptions of virtual machines. Of crucial importance is which subsets, if any, of the description can also be viewed as virtual machines. For example, I could have described the depth-first search program (including all of the abstractions pictured in Figure 3), as a single main program of several thousand lines of C++ code, i.e., without any function

or data abstractions, defining only a single virtual machine. The programmer therefore has considerable choice concerning the abstractions and description languages of the programs that they write.

The question in terms of program understanding is how the choice of abstractions and language affect the representations and processes of a program reader? I make the following hypotheses. First, program readers are able to interpret and understand programs at a single level of virtual machine description without respect to any of the other virtual machine descriptions. As a corollary, program simulation of a virtual machine is an inference strategy that effectively scales as programs grow in size. An additional corollary is that knowledge of the application domain of the program is often irrelevant to understanding lower level virtual machine descriptions. Second, a program understander constructs a multi-level representation of a program that reflects the virtual machine hierarchy of the program. This representation thus relates the different virtual machine levels to one another. Third, programmers are not constrained to build this representation in either a bottom-up or top-down fashion. Instead, programmers appear to choose the strategy that maximizes cognitive efficiency relative to their knowledge, the task, and their estimates of the relative costs and benefits of their strategic choices.

7 Understanding at a single level

In order to understand a description at a single virtual machine level, it is necessary for the understander to be able to accurately interpret the expressions of the description language. That is, the understander needs to possess some form of interpreter for this language in their mind. One way in which the semantics of an expression can be constructed is in terms of its implementation at lower levels of description. For example, the stack `Pop` operation can be understood by looking at the actual code that implements this at the next lower level. Of course, this may require looking at deeper levels of implementation, continuing until we reach primitives that we feel reasonably certain we know how to interpret.

But there are other ways to develop beliefs about linguistic expressions. If not, then what justifies stopping the process of looking at lower levels when we reach the level of C++ (or Ada or Lisp, etc.) primitives? Most experienced programmers have become so used to programming a virtual C++ (or Ada or Lisp) machine, that the primitives of these high-level languages feel like "the bottom", something solid and known. But how did we develop our interpretations of these primitive expressions? Most of us did *not* learn by reading the object code generated by the C++ compiler! And if we did, how much further down would we have to go in order to feel absolutely certain that we understood the terms of the description language of the object code – machine language, microcode, ... electrons?

We are able to learn the primitives of programming languages in a number of non-reductionist ways – reading informal, natural language descriptions of

the semantics, running short programs on different inputs, transferring meaning from other domains that share similar linguistic expressions, e.g., `read, write, until, +, =`. In an even more sophisticated fashion, we are sometimes able to discern the meaning of a term by recognizing its role in a referenced plan, such as the `Make_move` procedure from the Tic-Tac-Toe program.

But, if learning the primitives of a programming language does not require this reductionist process, then why should learning the primitives of any other virtual machine language? It might be that the primitives of current programming languages just happen to be precisely those kinds of virtual machine languages that are learnable without recourse to reductionistic reasoning, and that almost all other virtual machines require reduction to lower level primitives. But this is such an unlikely possibility as to be completely absurd.

I am not claiming that all ways of discerning meaning are equally accurate. For example, people make mistakes in transferring meaning from similarly named expressions from other domains, empirically demonstrated in [BS83, p10], and [New79, p21]. But I *am* claiming that we have several non-reductionist strategies that we use to understand, with varying degrees of confidence, the expressions of virtual machine languages.

7.1 Mental simulation as an inference strategy

In [Cur90], Curtis describes the extensive use of mental simulation as an inference strategy during the design phase of programming, i.e., before the implementation details of the code have been fully specified. In experiments involving protocol analysis on maintenance tasks of short programs, Pennington [Pen87a] and Littman, et al. [LPLS86] *both* report simulation being used extensively as an inference strategy. Ironically, they also both consider simulation as non-scalable to larger programs (although this is only implied in Littman's paper (p.97), but explicitly stated by Pennington (p.112)).

But the fallacy that simulation is non-scalable stems from the fact that effective simulation is not a function of the *aggregate* size of programs. The simulation of a virtual machine can be done *at the level of that machine*, without respect to how the machine is implemented. For example, mentally simulating the code required to completely implement the depth-first search program at the level of the C++ virtual machine is a daunting task. But simulating the 50 lines of C++ code in the algorithm presented above is quite simple, since neither the graph nor the stack implementations need to be simulated in terms of their implementation details. By analogy, consider a program that simulates the movements of the hands of a clock – one can, in fact, buy watches with virtual hands. The mechanical hands can be simulated without likewise simulating the mechanical devices, the ratcheting gears and posts, the jeweled pivots and pendula, that cause the hands to move as they do.

7.2 The importance of application domain knowledge

Ultimately, programs must pertain to the real world – programs that sort employee records, calculate compound interest for bank accounts, control the movements of a robot arm. In a Darwinian sense, programs that do not enable people leverage on problems in task domains become extinct.

There is empirical research supporting claims about the importance of real-world domain knowledge in designing and understanding programs for problems within that domain. Pennington [Pen87a] describes how programmers at high skill levels alternate judiciously between real-world and programming knowledge in understanding programs. In [Cur90, pxxxix], Curtis concludes from a number of his experiments that "Dominant designers were characterized by an unusually deep understanding of the application domain coupled with the ability to translate application behavior into computational structures."

However, if people are able to understand programs at one level without recourse to lower levels of implementation, it must likewise be the case that the lower levels are implemented without regard to how the expressions that they define are to be used at higher levels. For example the implementation of the stack used by the depth-first search program can be done without regard to its use in the depth-first problem.

This implies that knowledge of the application domain represented at the highest virtual machine level need only be of use in understanding the highest level virtual machine. In some sense, the application domain of the stack implementation *is* the stack functionality that it implements. Each level of description screens lower levels from the terms, and hence the referents, of higher level description languages. For constructing programs, this modularity property is extremely useful, for it permits different parts of programs to be constructed by different people at different times, places, and locations, using different languages, with only the functional requirements of each part needing to be communicated. By the same token, the knowledge required to *understand* virtual machines is modularized.

This does not contradict the cited empirical evidence concerning the importance of domain knowledge. For example, Pennington's experiments [Pen87b] are on short programs (30 to 200 lines), where it is unlikely that there is more than one virtual machine level. Likewise, Curtis [Cur90] is describing various aspects of the software development lifecycle, particularly the design phase, where the designer is making choices about how to decompose the problem into a virtual machine and language hierarchy. But nowhere does this research suggest that during all phases and for all people involved in constructing (or reconstructing the meaning of) a large programming project is knowledge of the application domain required in order to function effectively. Were it the case that domain knowledge is required in order to understand any of the lower level details of a program, this would render completely useless any of the complexity-reducing advantages of abstraction.

8 Multiple levels versus two levels

In [Pen87b, Pen87a], Pennington proposes that at least the following two different mental representations of the program are built:

> a representation that highlights procedural program relations in the language of programs, that we will refer to as the *program model* ... and a representation that highlights functional relations between program parts that is expressed in the language of the domain world objects, that we will refer to as the *domain model*. ... Effective comprehension also requires that the two (mental) representations be cross-referenced in a way that connects program parts to domain functions.

Pennington cites the two-level model developed by van Dijk and Kintsch [VDK83] for natural language understanding as the basis for her model. Bergantz and Haskell [BH91] argue in support of this two level model.

The strength of this two-level theory is that it is incontrovertible – but not because of Pennington's empirical validation. The two-level theory, at its essence, states simply that programs are representations, i.e., they have syntax and they denote. To deny this would be to deny that people reason about both the expressions of the programming language and the denotations of these expressions in terms of application domains. The billions of dollars of economic activity generated by the production and use of software argue otherwise.

The weakness with this theory is that "the language of programs" may not be one language, but a number of levels of description, each of which is associated with a virtual machine. What is viewed as a functional object at one level is a program object at the level above. For example, the language of stacks (e.g., push, pop, top, empty) is part of the domain model when considered from the point of view of the stack implementation. However, these same stack operations are part of the program model when considered from the point of view of the depth-first search algorithm. The same terms (push, pop, top, empty) thus serve both as program and domain elements, depending upon the level at which reasoning occurs.

As another case, consider a Scheme interpreter written in Scheme, where the program and domain languages are both Scheme.

But then it is only the context and intent of an utterance by a person about a program, and not the language of the utterance itself that determines whether a piece of program syntax is being reasoned about at the program level, the domain level, or both, for that particular utterance. It is thus impossible to classify statements by programmers during verbal protocols as to whether they are at the function or the program level. For example, according to Pennington's classification [Pen87a, p105], what makes "read in the cable file" procedural, i.e., at the program level and "computing area for cable accesses" functional, i.e., at the domain level? Both name operations on data objects. And both can be viewed as primarily syntactic, as a procedural statement, *and* as referential, as denoting an implemented function. The level of description language alone is

not sufficient to warrant different classifications for these nor many, if not most, other utterances.

We can trace the history of this multi-level view in the psychology of programming literature at least to Brooks, who clearly articulates that programmers construct multiple-level representations of programs [Bro83, p544]:

> The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain. ... Comprehending a program involves reconstructing part or all of these mappings.

My elaboration is that several of these levels of mapping are encoded in the language of the program itself, and that the representation of a program reader reflects the explicit hierarchical program structure.

In artificial intelligence, these ideas go back further, best articulated in Amarel's elegant description of the relationship between the level of problem description and the ability to find solutions to the encoded problem, providing a detailed example using different encodings of the Missionaries and Cannibals problem [Ama68]. Amarel's description has inspired a number of subsequent efforts by researchers in artificial intelligence to automate the process of change of representation through several levels of abstraction [Sac74, Ben90, EKM90, Kno90, Kno89, GW89] including efforts by the author of this paper [YTW96, AKPT91, YT90, KTY90, Ten90, Ten89, Ten87, Ten86].

One can even trace these ideas of multi-leveled representations through the history of mathematics to the work of Galileo and Descartes, as Haugeland does in [Hau85], by demonstrating that Galileo's use of line segments to represent elapsed times, and Descartes's use of algebraic symbols to represent spatial features, facilitates efficient inferences that preserve truth with respect to the original domain without manipulating symbols that directly denote domain objects.

My argument against a simplistic two-level view is virtually identical to the argument leveled against the two-level structure/function split that characterized the early Machine Functionalists in philosophy. Lycan [Lyc90, p60] summarizes[2]:

> Machine Functionalism's two-leveled picture of human psychobiology is unbiological in the extreme. Neither living things nor even computers themselves are split into a purely "structural" level of biological/physiochemical description and any one "abstract" computational level of machine/psychological description. Rather, they are all hierarchically organized at many levels, each level "abstract" with respect to those beneath it but "structural" or concrete as it realizes those levels above it. The "functional"/"structural" or "software"/"hardware" distinction is entirely relative to one's chosen level of organization.

[2]Thanks to Lyle Zynda, Professor of Philosophy at Indiana University South Bend, for seeing this connection to functionalism in philosophy, and for providing the reference.

To view either the program or the domain level as being described by a monolithic language is to oversimplify the power of programming languages to represent, and to miss the subtlety and craft of programming as it has evolved.

9 Top-down versus bottom-up inference

Given the hierarchical nature of programs, the question arises as to the order in which programmers read, understand, and construct their representations of the different levels of program description. For simplicity, I conflate these three activities (and other similar ones, such as hypothesizing, recognizing, predicting) under the rubric "inference", with the assumption that the order in which, for instance, a program is read will also be the order in which the program is represented and understood. I will use the phrase "inference strategy" to denote the order in which different parts of the program are read (understood, represented, etc.).

Since programmers are able to understand programs at a single level, there is in principle no constraint precluding any particular inference strategy. The two ends of the inference strategy spectrum that are discussed in the psychology of programming literature are termed *bottom-up* and *top-down*. The bottom-up view, as argued by Schneiderman and Mayer [SM79], Pennington [Pen87a], and Basili and Mills [BM82] is that programmers infer increasingly higher levels of program functionality by reasoning about lower level program chunks, starting at the lowest level of source code. This process is painstakingly detailed by Basili and Mills [BM82] for a short Fortran program, and presumably "this bottom-up process is typical in maintaining programs" (p282).

The top-down view, as argued by Brooks [Bro83], Littman et al, [LPLS86], Gellenbeck and Cook [GC91], and Wiedenbeck [Wie91], is that programmers reason using plan-like prior knowledge to guide the understanding process starting at the highest levels of description and moving through increasingly lower levels.

Another way to conceptualize this distinction, is to view the bottom-up approaches as being primarily data-driven, where the data driving the inferences is the actual code observed. Alternatively, the top-down approaches are primarily model-driven, where the models are knowledge structures retrieved from memory.

In this section, I argue that rather than looking for which inference strategy predominates, that we instead seek a theory, and associated empirical support, that describes the *factors* influencing the choice of inference strategy made by a program reader. Such a theory should be sufficiently detailed to enable us to manipulate these factors in specific ways, from which we should be able to observe predicted inference strategies, and account for differences in inference strategy already reported in the literature. Our claims about inference strategy should thus be relativized to the factors that are present and measurable. I additionally argue that one important such factor is the set of hierarchical abstractions and associated description languages that the original programmer

```
int test_for_win (board_type Board, player_type Player)
{
  int rw, riw, cw, ciw, dw;

  rw = 0;
  for ( int i = 0; i < SIZE && rw == 0; i++ )
    {
      riw = 1;
      for ( int j = 0; j < SIZE && riw == 1; j++ )
        if (  Board [ i ] [ j ] != Player )
          riw = 0;
      rw = rw || riw;
    }

  .
  .
  .

  return( rw || cw || dw );

}
```

Figure 5: Flat-structured code for testing a win in tic-tac-toe

has chosen to structure their program.

I will again illustrate these points with an example. Consider the code fragments in Figure 5 and Figure 6, that show different ways of testing for a win in a tic-tac-toe game, one flat, and with identifiers that do not easily suggest the function of the different program statements, and the other hierarchical, with semantically rich identifier names. The main point is not which one of these is easiest to understand; rather, it concerns what is the reading and inference strategy that a reader employs in trying to understand each of these code fragments. My claim is that the flat-structured code is considerably more difficult to read top-down, that is, in a model-driven, predictive fashion, because I have not provided much linguistic structure for doing so. In this case, a reader is virtually forced to inspect each bit of code in order to try to infer what functionality is associated with these lines of code, and to try to match this with their internal model of what is required in order to win at tic-tac-toe. Alternatively, the hierarchical code enables a top-down inference strategy, since the functionality of code sequences can be encoded using a single identifier, such as `row_win` or `column_win`.

Empirical research that attempts only to establish *the* predominant inference strategy without also searching for the conditions that give rise to these strategies – including conditions lying *outside* the reader's mind – can easily lead to endless skirmishing back and forth, when all parties may be correct

```
int test_for_win (board_type Board, player_type Player)
{
  return ( row_win ( Board, Player )    ||
           column_win ( Board, Player ) ||
           diagonal_win ( Board, Player ) );
}

int row_win ( board_type Board, player_type Player )
{
  for ( int i = 0; i < SIZE; i++ )
    if ( row_i_win ( Board, Player, i ) )
      return ( 1 );
  return ( 0 );
}

int row_i_win ( board_type Board, player_type Player, int i)
{
  for ( int j = 0; j < SIZE; j++ )
    if (  Board [ i ] [ j ] != Player )
      return ( 0 );
  return ( 1 );
}
```

Figure 6: Hierarchically-structured code for testing a win in tic-tac-toe

(and at the same time incorrect) due to factors that neither was aware of nor controlled for. For example, unconditional statements made by Pennington: "Our evidence to date suggests that the program model is constructed prior to the domain model" [Pen87a, p102], and by Koenemann and Robertson: "Programmers ... use bottom-up comprehension only for directly relevant code and in cases of missing, insufficient, or failing hypotheses" [KR91, p125] contradict one another without pointing to the reason for this discrepancy. But these unqualified statements are as much about the relative frequencies with which certain kinds of strategy-influencing factors exist in the code that is being read as it is about the cognitive strategies themselves. That is, Pennington is also implicitly stating that programs tend to be of the sort such that constructing a mental program model prior to constructing a domain model is an economical inference strategy; for if this were not the case, then why would programmers in general, tend to employ such a strategy? Similarly for the claim by Koenemann and Robertson. However, the lab-based testing methodologies used in both studies are insufficient for making claims about the relative frequency with which certain kinds of code exist in the world.

An adequate theory of directionality of inference and the construction of the meaning of programs must therefore account for why we observe a range of strategic behaviors. Such a theory should specify those factors influencing strategic choices that lead to cognitive efficiencies. At a coarse level, there is considerable evidence that these factors include the presence of domain and programming plans in the reader's mind [SAE88], as well as the reader's specific task [KR91]. At a more fine-grained level, they likely include: the agent's confidence in the match between a plan and a program fragment, the expected gain from using a plan for prediction, the expected cost of using an inappropriate plan and the associated costs of hypothesis revision, and the expected costs and benefits of hypothesizing about program function based on reading individual lines of code.

Clearly, many of these specific facotrs relate to objects *outside* the reader's mind, namely, the code itself. For example, top-down reasoning is facilitated in the hierarchical tic-tac-toe fragment presented above through its decomposition of functionality, captured through the use of semantically denoting names.

Even with the identification of only a few of these factors, it is possible to make conjectures about the observation of different strategies in research to date. For example, the small, flat-structured programs used in Pennington [Pen87b, Pen87a] and Basili and Mills [BM82] likely do not provide sufficient knowledge for model-driven reasoning to occur.

Conversely, it is likely that the top-down, as-needed reading behavior observed by Koenemann and Robertson [KR91] is only possible given hierarchically structured code such as that observed by the subjects of this study. These top-down strategies appear to offer such significant efficiencies on the specified tasks that these authors report their universal employment: "Subjects uniformly exhibited a focussed top-down process" [KR91, p127]. Similarly, it is likely that the difference in performance observed by Littman, et al [LPLS86] between subjects using systematic reading strategies as opposed to those using

as-needed strategies may actually have more to do with the directionality of inference that subjects from each group employed. That is, based on the protocol fragments provided for two contrasting inference strategies, only the more accurate, systematic reasoner also employed a top-down inference strategy.

The possible economies of using top-down processes are plausible if one considers the relative sizes of the immense space of possible plan/model hypotheses that might be associated with a fragment of code – the mental task faced by a bottom-up reasoner, versus the smaller (although still combinatorial) space of possible code fragments in the observable code that might be associated with an already referenced plan – the mental task faced by a top-down reasoner. Basili and Mills suggest the magnitude of this difference when they admit of their bottom-up efforts: "The authors would guess that it would take several weeks for a maintenance programmer versed in these concepts to develop and document an understanding of this program". Even recognizing that these authors were seeking a deep, formal understanding of the code, the example program contains only 61 non-comment lines of FORTRAN statements. In addition, as observed by Koeneamm and Robertson [KR91, p126-127], top-down strategies enable programmers to completely *ignore* parts of the program that they deem irrelevant to their given task. These authors state the following about the reading behavior of their 12 subjects performing a modification task on a well-structured PASCAL program containing 636 lines organized in 39 functions:

> We also found that a quarter of all procedures and functions were not looked at by *any* subject. Most of these were low level procedures like `max, min`, or `isdigit`, suggesting that subjects had common knowledge about the functionality of these procedures and, thus, saw no necessity to look at the code in detail or judged them as irrelevant. However, some of these code segments were indeed complex and important domain-specific produres.

As indicated by these authors, linguistic abstraction enables this ignoring. The function name matches a commonly used abstraction to an extent that the programmer views it as cost-*in*effective to look at the lower levels of code implementing this abstraction. This suggests that our technological and pedagogical efforts might be fruitfully directed toward supporting the development of code that can be correctly maintained through the use of top-down, as-needed strategies.

What emerges is a view that programs encode expertise about the structure of a problem and its solution obtained through mental effort expended by the original programmer. By externalizing this expertise using shared linguistic patterns, the original programmer makes it possible for subsequent programmers to understand the structure of the problem and its solution at only a fraction of the mental cost originally expended. To focus only on the cognitive aspects of programs within the mind of a single individual and to ignore the linguistic, communicative aspects of programming will yield only incomplete, unsatisfactory theories of meaning construction.

It is hoped that further empirical research will elaborate the theory of meaning construction so as to more fully account for:

- different programmers employing different strategies when reading the same program,
- a single programmer employing different strategies at different times in the same understanding episode,
- a single programmer employing different strategies when reading different programs that compute the same function.

10 Summary

My primary objective in writing this paper has been to discuss the implications of the use of hierarchical abstraction for program understanding, particularly as a means for interpreting experimental results. Computer scientists view the construction of abstractions and associated languages as a central task. By decomposing a program into encapsulated levels, the cognitive complexity of both the writing and reading of programs is reduced. Since programs can be understood at one level of description without regard to other levels, simulation remains a viable, scalable inference strategy, regardless of the aggregate size of a program. In addition, the higher levels of a program screen a program reader from having to consider the application domain in understanding the lower program levels. A multiple level mental representation of the program is hypothesized for the program reader, mirroring the hierarchical structure of the program itself; a two-level representation is simply a special case. Finally, the hierarchical, linguistic structure of a program is claimed to be an important influencing factor in the inference strategy that programmers use to read and understand programs.

References

[AKPT91] Allen, Kautz, Pelavin, and Tenenberg. *Reasoning about Plans*. Morgan Kaufmann, 1991.

[Ama68] Saul Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press, 1968.

[Ben90] P. Benjamin, editor. *Change of Representation and Inductive Bias*. Kluwer Academic Publishers, 1990.

[BH91] D. Bergantz and J. Hassell. Information relationships in prolog programs: do programmers comprehend functionality? *International Journal of Man-Machine Studies*, 35:313–328, 1991.

[BM82] Basili and Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, 8(3):270–283, 1982.

[Bro83] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[BS83] J. Bonar and E. Soloway. Uncovering principles of novice programming. In *Tenth symposium on the Principles of Programming Languages*, 1983.

[Cur90] B. Curtis. Empirical studies of the software design process. In *Human-Computer Interation – INTERACT '90*, pages xxxv–xl, 1990.

[Die94] E. Dietrich. Introduction. In E. Dietrich, editor, *Thinking computers and virtual persons : essays on the intentionality of machine*. Academic Press, 1994.

[EKM90] T. Ellman, R. Keller, and J. Mostow, editors. *Working Notes of the Automatic Generation of Approximations and Abstractions Workshop*. American Association for Artificial Intelligence, 1990.

[FWH92] D. Friedman, M. Wand, and C. Haynes. *Essentials of programming languages*. McGraw Hill, 1992.

[GC91] E. Gellenbeck and C. Cook. An investigation of procedure and variable names as beacons during program comprehension. In Koenemann-Belliveau, Moher, and Robertson, editors, *Empirical studies of programmers, fourth workshop*. 1991.

[GM93] J. Goodman and K. Miller. *A programmer's view of computer architecture*. Oxford University Press, 1993.

[GW89] Fausto Giunchiglia and Toby Walsh. Abstract theorem proving. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 1989.

[Hau85] J. Haugeland. *Artificial intelligence : the very idea*. MIT Press, 1985.

[Kno89] Craig A. Knoblock. Learning hierarchies of abstraction spaces. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 241–245, Los Altos, CA, 1989. Morgan Kaufmann.

[Kno90] Craig A. Knoblock. Learning effective abstraction hierarchies for problem solving. In *Proceedings of Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[KR91] J. Koenemann and S. Robertson. Expert problem solving strategies for program comprehension. In *ACM Human Factors in Computing Systems CHI'91*, pages 125–130, 1991.

[KTY90] Craig Knoblock, Josh Tenenberg, and Qiang Yang. A spectrum of abstraction hierarchies. AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions, 1990.

[LPLS86] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Plans in programming: Definition, demonstration, and development. In Soloway and Iyengar, editors, *Empirical studies of programmers*. 1986.

[Lyc90] W. Lycan. Introduction to part ii: Homuncular functionalism and other teleological theories. In *Mind and cognition: a reader*. Blackwell Publishers, 1990.

[New79] P. Newsted. Flowchart-free approach to documentation. *Journal of Systems Management*, 30:18–21, 1979.

[Pen87a] N. Pennington. Comprehension strategies in programming. In Olson, Sheppard, and Soloway, editors, *Empirical studies of programmers, second workshop*. 1987.

[Pen87b] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[Sac74] Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[SAE88] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the comprehension of computer programs. In Chi, Glaser, and Farr, editors, *The nature of expertise*. Erlbaum, 1988.

[SF89] G. Springer and D. Friedman. *Scheme and the art of programming*. McGraw Hill, 1989.

[SM79] B. Schneiderman and R. Mayer. Syntactic/semantic interactions in programming behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, 8:219–238, 1979.

[Tan90] A. Tanenbaum. *Structured computer organization, 3rd edition*. Prentice Hall, 1990.

[Ten86] J. Tenenberg. Planning with abstraction. In *Proceedings of the 5th National Conference on Artificial Intelligence*, 1986.

[Ten87] J. Tenenberg. Preserving consistency across abstraction mappings. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, 1987.

[Ten89] J. Tenenberg. Inheritance in automated planning. In *First International Conference on Principles of Representation and Reasoning*, 1989.

[Ten90] J. Tenenberg. Abstracting first-order theories. In P. Benjamin, editor, *Change of Representation and Inductive Bias*. Kluwer Academic Publishers, 1990.

[VDK83] T. Van Dijk and W. Kintsch. *Strategies of discourse comprehension*. Academic Press, 1983.

[Wie91] S. Wiedenbeck. The initial stage of program comprehension. *International Journal of Man-Machine Studies*, 35:517–540, 1991.

[YT90] Qiang Yang and Josh Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 204–209, Boston, MA, August 1990.

[YTW96] Yang, Tenenberg, and Woods. On the implementation and evaluation of ABTWEAK. *Computational Intelligence*, 12, 1996. to appear.