The Semantics of "Semantics"

Charles Petrie • *Stanford University*

Outside of computer science, semantics is the providence of philosophy, where we talk about what we mean when we talk, as well as ontology (what there is to know) and epistemology (how we know it). The nice thing about computer science is that, in contrast to philosophy, we can establish whether different representations make a computational difference. That's what makes what we do engineering/science.

It's All Just Syntax, Isn't It?

That's what skeptics say when we try to apply the concept of semantics to computer science. One reason they do, at the risk of building a strawman argument, is that people often throw the word "semantics" around in computer science loosely these days, often denoting only a particular syntax. In particular, discussion of the Semantic Web often refers only to using either RDF or OWL-S. As I wrote in 1998 in response to much published hyperpole, XML wasn't going to save the world because it was only a syntax.[1] It lets us encode some semantics, but it doesn't provide them. So, I'm sympathetic to the syntax question to a certain extent. But it's not the end of the story.

I'm not fond of using Wikipedia as an authority, but it's useful to read the description there of semantics in computer science. The usual meaning of "semantics" as intended in the discussion of the Semantic Web and Semantic Web services (SWS) is that of "axiomatic semantics."

Two years ago, Martin Hepp wrote in this space about the different meanings of "ontologies" in different camps ("Possible Ontologies: How Reality Constrains the Development of Relevant Ontologies," Jan./Feb. 2007). However, if we're really talking about the Semantic Web and SWS, I insist upon nailing this down to axiomatic semantics. Otherwise, our discussion is ambiguous and thus unscientific (C. Petrie, "No Science without Semantics," Jul./Aug. 2007). How can we be doing computer science if we're using fuzzy concepts that don't make a computational difference?

The idea of axiomatic semantics is that we constrain the use of a vocabulary (a set of distinguished terms) not only by the hierarchical relationships among the terms in a taxonomy but also by axioms that tightly constrain the legal inference using these terms. A formal ontology is a set of such terms — that is, a taxonomy and axioms (which might include inheritance of properties within the taxonomy).

When using formal inference with a formal ontology, any inference consistent with the axioms is okay. The whole set of possible consistent ramifications and interpretations comprises the ontology's semantics. The trick is often to interpret the relations used in such an ontology.

More Semantics Is Better for Responding to Change

Is it still all syntax? Well, ultimately yes, but this is the wrong question to be asking. The right question to be asking is whether our representation makes a computational difference. And when we constrain the computation to be consistent with the axioms, we make a difference. The more tightly constrained the use of the terms, the more we have formal semantics for them.

Various frameworks and schemas can provide semantics. An article by Amit Sheth, Cartic Ramakrishnan, and Christopher Thomas[2] can help us begin understanding that XML Schema offers only a starting point for representing semantics, and description logics offer more. A W3C specification such as OWL-S builds on top of RDF, which is built on XML, but other options are available, and, of course, first and higher-order logics let us express more.

This isn't the end of the story, either. Now we must determine what to express. In any par-

 1089-7801/09/$26.00 © 2009 IEEE

ticular application, is it useful to have so constrained the terms? Are they constrained enough? This is the heart of knowledge engineering: finding the right representation. But how do we know?

I've argued before in this column that the proper way to view (formal) semantics is as an advanced software engineering technology ("It's the Programming, Stupid," May/June 2006). Anything you can do in semantics once you can do just as well in Java programming. A start to knowing if you have good semantics is to see whether you've solved the problem via testing. Then, you can find out whether you've done at least as well as someone hard-coding the solution.

But semantics should be more flexible, or it has no obvious semantics over traditional software engineering techniques. A good C++ programmer will always beat someone using complicated ontologies and logic in building a specific program, one time. But for some classes of complex programs — and changes to which a programmer has to adapt — we should be able show that semantics has an advantage, yes?

I should explain that many semantics proponents (and I am one) say that semantics permits or facilitates reuse and interoperability. My view is that this amounts to less programming in response to change. If my system can talk with yours or repurpose an ontology with little if any change, then this means that less programming was required to adapt to changing conditions. That's the general principle.

Advocates will also note that we can use semantics to answer queries that simple keyword searches can't. For instance, what is the oldest Western university not founded by monks? Or, to take an example from the Semantic Web Services Challenge (SWSC; http://sws-challenge.org) set of shipping and discovery scenarios, what shipper, or combination of shippers, is required to move my 25-lb package from New York City to Tashkent by 5 p.m. local time Tuesday?

Again, a programmer could write a program that would find the appropriate shipper (or university), perhaps even screen-scraping from Web sites. And that programmer could write a new program for some other kind of logistics task. But the hope is that the programmer could reuse the semantics for time and location, at least, for this new program. Again, the general principle is that semantics should be a superior software engineering technique. In the extreme case, no new programming should be needed, as is the case with keyword search today, but this particular hope should be tempered.

Some Known Hard Issues

A common vision is that all Web applications will use open semantics (perhaps derived from less semantic sources with some kind of Web 2.0 methodology). Some of us don't believe in this vision, not because it's too hard to do but because we doubt that the same representation can be used for all purposes. This is the import that I take from Drew McDermott's "Critique of Pure Reason."[3]

It's also what some of us take from the Noy and McGuinness methodology of ontology development.[4] Munindar Singh especially has commented on this.[5] All this doesn't mean that semantics can't be useful as a software engineering technique — only that we should remain skeptical of claims of universal interoperability and reuse. However, within some restricted domains, semantics could prove very useful.

Christoph Bussler and I expect that such domains might well be industrial — that is, occuring in "industrial service parks," with some interoperability in the future ("The Myth of Open Web Services," May/June 2008). This does beg the question, though, for what set of problems might a common representation be useful? Let me duck that one by saying that we can usually surmise informally what those might be and confirm this with some testing. A formal answer awaits those who would like to research this meta-problem.

Trading One Issue for Another

I got rid of the problem of defining semantics as opposed to syntax by saying it's a continuum that improves software engineering for a large, complex set of programming problems — in particular, large systems that have to adapt to change. Is this a testable hypothesis?

At this point, I must confess the short answer is "not yet." Should you be shocked, let me explain why, and why this isn't such a bad result.

Let me go back to the subject of "ontologies" for a moment. There is, at least in the AI community, a notion of modeling that's different from that of the software engineering community. In the latter, it's typically the software system that's modeled, not the task domain per se. When you want to change the program, to adapt to new circumstances, you change the software model.

In the AI community, we model a task domain more or less directly in logic, using an ontology. Then, we execute the logic using some computational logic language, possibly compiling this program to achieve efficiency. When we want to adapt to new circumstances, we might have to somehow improve our domain model, but nothing should change with respect to what we'd usually call code.

In such computational logic, distinguishing between the declarative statements and the rules that use them is easy. Then, we could say whether a change in a program, in order to adapt to a change, necessitates simply adding more statements, changing the existing statements,

or changing the logic model's rules. This would be a good indication of the representation's adaptability. We can imagine comparing different computational logic models to one another with respect to changes in the model.

We can't use such distinctions when comparing the logic-based technique to other software engineering techniques.

In the software engineering community, some kind of programming model, less general than logic, is often specialized by the programmer to create a domain-specific model. UML is such a programming model: it can be used for a specific problem and then compiled into an executable.

Such a model might look nothing like logical rules and statements. Now, what is data and what are rules? And was the model harder to change? What if it is graphical and you just move around a few icons and recompile? Can we even say that it isn't "semantic"?

So, we might try to move to a more general measuring technique. We did this in the SWSC. We've resorted to the "surprise problem" methodolgy. If you've solved a public problem, you're invited to solve a new version of that problem. We freeze your existing code and give you the new version a short time before the workshop. In the workshop, we evaluate whether you managed to solve the new version. (We also look at your new code, comparing it to the old, but this isn't part of the formal metric.)

So, now we have a method for determining how semantic your code is by how well you responded to the fast change challenge. Oh, wait. If you're a really great Java programmer, maybe you can just write a new program solving the new problem faster than I can change my logic-based program. The equivalence class of programming techniques able to quickly adapt to a change might depend heavily on the programmer.

I don't like time-based tests. I would rather have a more objective measurement of code change, but our SWSC community hasn't been able to discover a common one. At least we're timing the programmer rather than the code execution time. These days, execution performance is overrated when you realize that increasingly complex and interconnected programs require a human programming cost that might not scale at all. But timing programmers is a poor technique — it's just that, like democracy, it seems to be the only way forward right now.

The good news is that at least we've reduced the problem to a general software engineering one. If we can say anything about whether one programming technique is better than another with regard to adaptability, then we ought to be able to determine whether, say, logic-based programming is better.

The not-so-bad news is that the problems that we've posed so far in the SWSC are probably not yet sufficiently complex, or perhaps semantic. We likely need to expand the shipping and discovery scenarios and indeed the whole set of supply-chain scenarios. Building up such problems, building programs that solve them, and testing changes could be so difficult that we probably won't get around to doing this for a long time.

That doesn't mean we won't. We'll keep trying as long as we can sustain the effort. And as enterprises form large interlocking programs running over the Internet that must adapt to change, it's increasingly likely that semantics will be the only software engineering technique that will work. That is, the richer the semantics (though possibly in a lightweight, less-expressive framework), the more likely that programming can scale to meet the need for flexibility.

The bottom line is that I claim that the only proper measurement of semantics is as a software engineering technique, and semantic technologies should be evaluated as such, and against all software engineering techniques, if our community is to have any credibility. The only way to do this is via some method that tests the difficulty of modifying applications in response to changing conditions or requirements, which might include the need to interoperate with other applications. If we don't test such adaptability, then the evaluation is less meaningful, especially to the community skeptical of semantics.

References

1. C. Petrie, "The XML Files," *IEEE Internet Computing*, vol. 2, no. 3, 1998; www-cdr.stanford.edu/~petrie/online/v2i3-webword.html.
2. A. Sheth, C. Ramakrishnan, and C. Thomas, "Semantics for the Semantic Web: The Implicit, the Formal, and the Powerful," *Int'l J. Semantic Web & Information Systems*, vol. 1, no. 1, 2005, p. 18; http://knoesis.wright.edu/library/download/SRT05-IJ-SW-IS.pdf.
3. D. McDermott, "A Critique of Pure Reason," *Computational Intelligence*, vol. 3, no. 33, 1987, pp. 151–160.
4. N.F. Noy and D.L. McGuinness, *Ontology Development 101: A Guide to Creating Your First Ontology*, tech. report, Stanford Univ.; http://protege.stanford.edu/publications/ontology_development/ontology101.pdf.
5. M.P. Singh, "Tools for Pragmatics: Metadata for Nothing; Ontologies for Free?" www.csc.ncsu.edu/faculty/mpsingh/papers/positions/Singh-meaning.pdf.

Charles Petrie has been a senior research scientist at Stanford University since 1993. His research interests include concurrent engineering, virtual enterprise management, and collective work. Petrie has a PhD in computer science from the University of Texas at Austin. He is EIC emeritus and a member of *IC*'s editorial board. Contact him at petrie@stanford.edu.