Call by Meaning

Hesam Samimi, Chris Deaton, Yoshiki Ohshima,
Allesandro Warth, Todd Millstein

To appear in ACM International Symposium
on New ideas, New Paradigms, and Reflections
on Programming & Software, Onward! 2014.

VPRI Technical Report TR-2014-003

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

Call by Meaning

Hesam Samimi
Communications Design Group,
SAP Labs
hesam@ucla.edu

Chris Deaton
Cycorp
cdeaton@cyc.com

Yoshiki Ohshima
Viewpoints Research Institute
yoshiki@vpri.org

Alessandro Warth
Communications Design Group, SAP Labs
alex.warth@sap.com

Todd Millstein
University of California, Los Angeles
todd@cs.ucla.edu

Abstract

Software development involves stitching existing components together. These data/service components are usually not well understood, as they are made by others and often obtained from somewhere on the Internet. This makes software development a daunting challenge, requiring programmers to manually discover the resources they need, understand their capabilities, adapt these resources to their needs, and update the system as external components change.

Software researchers have long realized the problem why automation seems impossible: the lack of semantic "understanding" on the part of the machine about those components. A multitude of solutions have been proposed under the umbrella term *Semantic Web* (SW), in which semantic *markup* of the components with concepts from semantic ontologies and the ability to invoke queries over those concepts enables a form of automated discovery and mediation among software services.

On another front, programming languages rarely provide mechanisms for anchoring objects/data to real-world concepts. Inspired by the aspirations of SW, in this paper we reformulate its visions from the perspective of a programming model, i.e., that components themselves should be able to interact using semantic ontologies, rather than having a separate markup language and composition platform. In the vision, a rich specification language and common sense knowledge base over real-world concepts serves as a *lingua franca* to describe software components. Components can query the system to automatically (1) discover other components that provide needed functionality/data (2) discover the appropriate API within that component in order to obtain what is intended, and even (3) implicitly interpret the provided data in the desired form independent of the form originally presented by the provider component.

By demonstrating a successful case of realization of this vision on a microexample, we hope to show how a programming languages (PL) approach to SW can be superior to existing engineered solutions, since the generality and expressiveness in the language can be harnessed, and encourage PL researchers to jump on the SW bandwagon.

Categories and Subject Descriptors D.2.12 [*Interoperability*]: Interface definition languages; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques

Keywords language design; program composition; automated discovery; specifications; meanings; semantic web

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2661136.2661152

1. Introduction

We build software out of existing components. Two reasons may be attributed to this phenomenon.

First, in order to make useful tools, developers have to build artifacts that work within common platforms, such as within a web browser, or in connection to a database backend. There are many parts that need to be in place to make that work, and so there is an incentive to utilize what is already a working part of the ecosystem.

Second, developers desire to make use of the plethora of components that are available on the Internet: libraries, web services, applications, etc. Such reuse allows programmers to save time by avoiding the need to reinvent the wheel, and it enables new kinds of networked systems and applications. We expect the level of connectedness to increase over time.

Unfortunately, the result is that software today is incredibly complex and fragile. In order to make a simple web app,

for instance, the programmer must first *discover* the many components that are needed: various frontends, databases, libraries, virtual machines, and so on. Once the components have been identified, the programmer must install and set them up appropriately and find out how they can be used. If a single component is not set up correctly, the system won't work. Even harder can be the task of enabling *data compatibility* among components, as interfaces and data units and formats may not be compatible out of the box.

Even when success is achieved after all of this arduous, error-prone, and manual burden, the resulting artifact remains incredibly fragile. If assumptions about the environment in which the software runs changes, e.g., one component undergoes a slight interface change or becomes unavailable, the entire system will grind to a halt.

We can examine the *offline* and *online* aspects of the *component discovery* and *data adaptation* problems. The current best practices to enable *offline* discovery is for the websites offering software libraries and services to include natural language descriptions of what is offered. The human programmer then manually searches with keywords on a search engine like Google to get a list of matches and then uses human judgment to decide what components to use. It is plain to see that the approach is not amenable to automation and scalability.

In the *online* context, we continue to rely upon referring to components by *name*: if one knows the name of the component needed, one can refer to and obtain it. This agreement on names is the basis of how we can refer to a variable in scope in a procedural language and what many publish-subscribe systems [7] use to perform matchmaking when forwarding published data to subscribers. The flaw, however, is that relying upon naming conventions does not scale beyond local applications. At the scale of an ecosystem as vast as the Internet, parties never find consensus on names [1].

Similarly, the current practice for *data adaptation* is to manually browse developer forums such as Stackoverflow to learn how to use a component or find some adaptor code to convert the data to the desired format. One may also attempt to write procedures to dynamically detect the received data formats and handle the task of conversion to the desired forms. Both approaches are fragile and non-scalable (i.e., not amenable to automation and infeasible to handle manually as the number of cases grows).

Our goal is to move away from these current practices and veer toward programming paradigms that automate and facilitate the tasks of *component* and *data discovery* and *data adaptation*. Moreover, if we take the fragility problem and the idea of networked applications seriously, these tasks must be done *online* (i.e., dynamically and continually), rather than *offline* (i.e., only once during development).

[1] An odd statement, as the Internet works based on names (DNS), but the reason is that all is routed through a central authority (ICANN).

We observe that the source of the problem preventing us from realizing these goals is that software components themselves have no real semantic "understanding" of one another, forever requiring manual human intervention to interpret and act. Thinkers in the field of AI such as Minsky [19] and Lenat [12] have long argued the importance of *common sense reasoning* in building intelligent systems. The idea is that in human interactions much more information is exchanged implicitly than explicitly, because of the vast, shared, everyday knowledge of the real world. For our purposes, we consider "common sense knowledge" to be any piece of knowledge widely shared by some community.

Suppose, for example, I would like to tell you about a film, but cannot seem to remember its *name*. Despite that, there are many ways I can still convey to you the film in question, as I can leverage your preexisting knowledge on the subject: actors in the film, the city where the story takes place, a synopsis of the film's plot, etc. Similarly, the exchanges among automated systems must leverage common sense to achieve true scalability and robustness.

Imagine that our computers were equipped with as much common sense and knowledge about the real world as the typical human. In that case, components could simply "talk" to each other, understand, and be understood. Because of the existing knowledge they have about the world they live in, they could make judgments, connections, and conclusions about what they hear from one another. The tasks of finding the right service or data component and adapting the data based on the individual's needs would become possible entirely dynamically and automatically.

In this paper, we present a multi-tiered vision for tackling the discovery and compatibility problems along these lines. The vision tiers get progressively more ambitious and is premised on the existence of a real-world ontology and specification language that can be universally adopted as a *lingua franca of meanings* to tie software to real-world concepts. We explore a programming model in which a theorem proving engine operating over that language of meanings is employed in order to bridge the understanding gap among components.

In the base tier of the vision, instead of relying on names to refer to what is sought (*access by name*), components can be annotated with specifications over this meanings language to advertise themselves and invoke queries over the same language to discover other available components. This is part of a process which we dub *call by meaning*.

In the second tier, components also advertise their own API and query over a discovered component's API, using the same meanings language. In this way, the customer of a service/data component not only discovers it using the meaning language but also discovers the right API within the component for achieving a given task.

In the third tier, components can remove their sensitivity to the formats and units of the data obtained from the

discovered component by relying on the theorem prover to semantically adapt the data based on their individual needs, e.g., converting the units, implicitly. In the fourth, most ambitious tier, all of the above occurs dynamically, as opposed to only statically as auto-suggestions provided by the IDE during development.

We report on our initial investigations related to this vision. To play around with these ideas we developed a prototype programming environment called $Nav\bar{a}$, in which we employ Cyc [25]—a common sense ontology, specification language, and theorem prover—as the meanings lingua franca. We also applied $Nav\bar{a}$ to partially address two scenarios posed in the Semantic Web Service Challenge (SWSC), a popular testbed for researchers aiming to address some of the challenges we have mentioned earlier.

The paper is organized as follows. Sec. 2 discusses the problem of flexibly composing software components. In Secs. 3 and 4 we overview the existing practices and discuss how they fall short. In Sec. 5 we discuss our vision for a programming model that can overcome the challenges. An account of our initial investigation toward realizing this vision is presented in Sec. 6. Our experiment with the SWSC is summarized in Sec. 7. We discuss the challenges for achieving our vision in Sec. 8. Finally we compare with related work and conclude.

2. Problem

We are interested in relieving the programmer of the burden of discovering data or service components and enabling compatibility between the two ends when exchanging data. Below we list several tiers of our vision to achieve the level of "loose coupling" we believe is needed for combating the problem in a truly automation-friendly and scalable manner.

***Tier I*: Component Discovery** We want to automatically discover a desired data or software component. To make this possible, the programmer must be able to describe what a desired component does or represents, i.e., "*I need a component that does X*" rather than "*I need that component*." Note that simply using names for this purpose does not scale. After all, components are developed independently and often in isolation, so a name is not a reliable identifier of a component.

***Tier II*: + API Discovery** Once we identify and obtain the desired component, how should we communicate with it? Knowing its *interface* (e.g., that the component responds to a message `foo()`) doesn't help unless we know what the methods are supposed to do. Therefore we need to discover "*How can I get this to do X?*"

***Tier III*: + Data Adaptation** At the scale of the Internet, one does not get to control at all times exactly how various information is represented, nor how often the format changes. A component must be able to interpret and adapt the data it receives from other components, no matter how that data was originally presented, i.e., "*What is the value of X in the format/units that I care about?*"

***Tier IV*: All Online** We may want to perform the above tasks *offline*, that is as suggestions automatically produced by the IDE as a development aid. Such an ability would already be a huge leap over the current practices. Yet it would still leave the problem of *fragility* of existing software in tact; APIs and formats in which data is represented evolve all the time and the execution environment may undergo changes, so something that works today may not work tomorrow. Thus, we need to achieve the above goals *online*, that is, the components do it autonomously and on a dynamic and continual basis.

The above vision is somewhat of a holy grail in software development, and any amount of progress, even only on the base tier, can profoundly improve the state of art.

3. The Semantic Web

Researchers have been pondering the grand vision of *the Internet of software services* and the automation of discovery, mediation, and composition among them, for several decades under the umbrella of the *Semantic Web* (SW). We believe that it is necessary for the programming languages community to join the Semantic Web efforts. Being born from outside of the language community, the Semantic Web solutions approach the problem as an "afterthought." That is, it is naturally their assumption that we have to work with existing programs written in existing programming languages, which obviously themselves lack any semantic IQ. Thus virtually all solutions are posed as semantic annotations written in *markup* languages such as OWL-S (Semantic Markup Language for Web Services) / WSDL-S (adding semantics to Web Services Description Language) to attach meanings to existing code [6, 20]. As a consequence, the integration of software components occurs at a stage that is entirely separated and outside of the programs themselves.

On the other hand, interesting possibilities arise when the notion of semantic discovery and composition is part of the software development process, and even part of the programming model itself. For example, an IDE can perform semantic search for desired components during development time, and programs themselves can reason semantically in order to dynamically interoperate with other components. Such an integration allows the discovery and composition mechanisms to leverage the generality of the programming language and allows the software itself to control this process, for example based on its runtime state. We take the first steps toward such a vision in this paper.

4. Manual Search / Access by Name

In programming languages, the current practice of obtaining and accessing a software entity *offline* (during development) works by manual search. For the *online* scenario, we rely on

agreement on names. Both practices clearly fail to achieve the goals presented in the previous section.

Consider an example of a GUI clock application shown in Figure 1, which displays a wall clock showing the current time. The application has two main components. The green object on the left is the source of time data, say *www.time.gov*, produced in some format, say number of seconds passed since 1970 (Unix *epoch* time). The clock hands on the right, representing the current hours, minutes, and seconds, are the consumers of that data. Given the current time they determine their rotation angle.

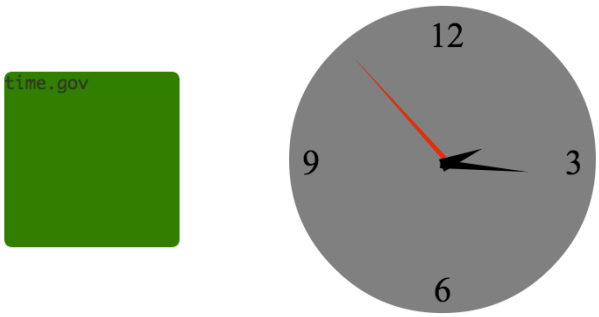

Figure 1. Clock example

For clarity, let us operate in the familiar JavaScript (JS) language. For the example in Figure 1, the source of time on the left is represented as a JS object with a single method `now`, returning the current time data:

```
clock = {
  now: function() {
    return Math.floor(
      new Date().getTime() / 1000)
  }
}
```

And thus the consumers of this component need to refer to it and access and invoke its method to obtain the needed information. Again we can consider both the offline and online scenarios. First, the developer searches on Stackoverflow "*how to get current time in JavaScript.*" Let us imagine she eventually comes across the code snippet above that does the trick, so she imports this code in her program. Now she is ready to use that component by referring to it by name:

```
var epoch = clock.now()
setAngleBasedOnEpoch(epoch)
```

The flaws are evident:

Problem of "Object Discovery" The consumer had to perform manual search to find the right component and use human judgment to evaluate the choices. The drawback is also apparent in the *online* view of the problem. Even if the component with the clock functionality is already in scope in the dynamic environment of the consumer component, the only way to "discover" it is by agreement on a name to look for (here `clock`). The problem with reliance on naming agreements isn't restricted to common programming languages. For example, in a common form of the publish-subscribe paradigm [7] the messages get routed from publishers to subscribers based on classification using *topic terms*. Again, the flaw is that agreement on topic names does not scale to the world at large.

Problem of "API Discovery" Once the `clock` component is found, how can we know with certainty that the information we want can be obtained by invoking that component's `now` method?

Problem of "Data Adaptation" Without inspecting and understanding the code, the consumer is taking a leap of faith in assuming that `now()` returns the current Unix epoch time. It's possible that the data comes in another format, say in string form `YYYY:MM:DD:HH:MM:SS`, or even a voice message reading the current time in Spanish!

In the next section, we share a vision for a programming model that can address these challenges and is founded on facilitating a form of real understanding between objects.

5. Call by Meaning

Let's do a thought experiment. Imagine our programming environment comprehends English and has as much common sense about the real world as the programmer. Assuming English as a common language of *meanings* we can now redo the clock example while avoiding the problems described above.

We will allow the producer of data to annotate it with a description in English. A reserved function named `spec` returning a string array can be used to make statements about the meaning of the object and some of its API:

```
clock = {
  now: /* same as before... */,
  spec: function() {
    return ["this represents a clock",
      "now represents the current time in terms
of seconds since 1970, i.e., Unix epoch time"]
  }
}
```

Here the underlined *this* and *now* appearing in the specifications are logical variables referring to the object being produced—i.e., `this`—and its method `now`, respectively. The component is stating that `this` represents a clock and it has a property called `now` that provides the Unix epoch time.

Given this description, consumers no longer need to assume knowledge of the name given to the component or any parts of its internal state. The programming environment can discover this component for them, given their needs in terms of meanings. They may also access various APIs *by meaning rather than by name*. Let us see how.

We add to our language the ability to query a common sense knowledge base and theorem prover K about a given object's specifications. If *KB* denotes K's entire common sense knowledge base about the world, S_o the meanings that object o has claimed by running its spec function, and Q the query asked about the object o, then

K.ask(o,Q)

denotes asking K the logical question "*Can you prove (KB* $\wedge$ S_o*) implies* Q*?*" Within the query Q, the argument object o can be referred to by the logical variable that.

Additionally, we give our system the ability to keep a repository of available objects which have been annotated with spec meanings. We then provide another query method

K.find(Q)

which denotes asking the system to find an object o in the repository for which K.ask(o,Q) returns a value with a truth value of true in JavaScript. In the case of multiple matches, find may return any one choice. The undefined value is returned when no matches are found.

5.1 Object Discovery

As an example, the call to find in the code line below discovers an object acting like a clock, should one exist in the environment.

```
var clock = K.find("is that capable of telling
the current time?")
```

Because our fantastical theorem prover K understands real-world concepts and knows a thing or two about clocks, it can deduce that the clock object in the repository is indeed capable of telling the current time. Thus the find call above will return the clock object, that the clock hand GUI objects can employ.

5.2 API Discovery

We also allow the query Q to mention a set of logical free variables $\overline{V}$, in which case the call to ask is interpreted as "*Can you find a* model*—i.e., a value binding for each of the variables in* $\overline{V}$*—such that (KB* $\wedge$ S_o*) implies* Q*?*" If the query cannot be satisfied (either because it is unsatisfiable or because K simply cannot prove either way), ask returns the value false. Otherwise ask returns a dictionary object representing the model that makes the query true.

For example, once we have discovered the clock object, we can invoke an ask call to discover the method in its API that would provide the current time data. The invocation to ask in the example below returns a model with a binding for methodName, which can then be referred to in the program, as done at the end of the expression of line 2.

```
1 if (clock) {
2   var epochMethodName = K.ask(clock, "what
  is the name of that's property (methodName)
  which represents the current time in Unix epoch
  format?").methodName
3   var epochMethod = clock[epochMethodName]
4   var epoch = epochMethod()
5   setAngleBasedOnEpoch(epoch)
6 }
```

From what the object has claimed about its property named now we can deduce that the property indeed represents the current time. Thus the return value from the ask query above is the model {methodName: "now"}. It follows that (clock[K.ask(clock, "...").methodName]) is indeed equivalent to the JS expression (clock["now"]), or equivalently (clock.now). Of course, the programming language can provide a syntactic sugar that combines lines 2-3 to ask the value of the queried property directly.

5.3 Dynamic Specs

You may have noticed that meaning specifications could possibly obviate the need for having or accessing object properties altogether. To demonstrate, consider in the code snippet below how an alternate version of the provider of the current time data does not say anything specifically about the property itself, yet it inserts the current data directly into the specifications:

```
1 clock = {
2   now: /* same as before... */,
3   spec: function() {
4     return ["this represents a clock",
5       "the current time in terms of seconds
  since 1970, i.e., Unix epoch time, is " + this.now()]
6   }
7 }
```

The expression (this.now()) in line 5 will cause the clock object to dynamically generate a specification containing the current Unix time. The example demonstrates a case where specifications are more tightly connected to the code, as they are constructed from the dynamic values of expressions. This contrasts with markup annotations in Semantic Web solutions, where specifications are added later and have no real connection to the program.

5.4 Data Adaptation

We haven't yet tried to address the data format sensitivity problem. The consumers are assuming that the producer of time has provided the data in terms of Unix epoch time. But we don't need to make this assumption! The information included as part of the clock object's specifications should be enough for our fantastical theorem prover and knowledge base to be able to give us the current time data in the format and units that suits us. So let's refactor the consumer code in Sec. 5.2 (assuming it's for the *seconds* clock hand):

```
1 if (clock) {
2   var secs = K.ask(clock, "what time is
  it in terms of seconds sec on a wall clock?").sec
3   setAngleBasedOnSec(secs)
4 }
```

As seen in the query on line 2, the object is directly asking for the value in terms of seconds on the wall clock rather than

having to deal with the epoch time. K uses its knowledge about units of date and time to implicitly do the conversion job from the received into the requested form.

We have now described the various tiers of our vision for an intelligent, networked programming environment. It would be liberating and revolutionary to have it. Certainly we do not have a real solution fully realizing this vision at any level. Nonetheless, in the next section we will share some promising results we obtained from our initial investigations into utilizing an existing real-world ontology and theorem prover towards this purpose.

6. Investigation

We have identified a real-world ontology and specification language which has its own powerful reasoning engine. We then developed a prototype programming language that supports the *call by meaning* paradigm by employing this ontology and language for meanings.

6.1 Real-World Specification Language and Reasoner

In order to have what we wished for in the previous section, we need a reasoner (which we called K) that not only comprehends real-world concepts and common sense, but it is also equipped with a powerful and effective theorem proving engine.

Over the last decade or so powerful theorem provers have come along. For example, SAT and Satisfiability Modulo Theories (SMT) solvers (e.g., [5]) have become a fixture in program analysis and verification tools. These theorem provers, however, work entirely on symbolic representations. They are powerful in making logical deductions yet lack any baked-in common sense knowledge, such as "*What is a clock?*," "*Can it tell the current time?*," etc. Therefore these solvers require the user to symbolically describe every piece of knowledge involved in reasoning over a particular query. This cannot help with our goal of bringing about understanding at a universal level.

On the other hand, under the term *Semantic Web* technologies, we have also seen a surge of interest in real-world ontologies and semantic markup languages to annotate software components. For example, The W3C Web Ontology Language (OWL) is a popular way to formally represent an ontology of things and relations among them. Because of the standardized format, such a knowledge base can be exploited by computer programs to facilitate a form of semantic understanding and analysis. Unfortunately, many have serious limitations. Some make semantic connections by associating a component with a term from a semantic ontology (e.g. `BuyDeltaTicket`, which is a kind of `BuyAirlineTicket` which is a kind of `Buy`), yet they provide no baked-in knowledge or axioms about those concepts. In many cases they aren't backed up with reasoning engines with comparable deductive powers, e.g., many are based on text/name matching, or else adopt logic-based reasoning which do not seem to scale well to again widespread applicability in practice [20].

Therefore, we chose to explore instead the use of a tool called Cyc [25] which seems to bridge the gap between expressiveness, powerful reasoning, and having vast built-in knowledge of the real world. Cyc—developed by Cycorp—is a formal language and common sense ontology that incorporates human-level semantic understanding of concepts and facts like "clock," "time," "now," "kidneys," "units," "humans need air," etc. The Cyc project started over 30 years ago and by now it contains millions of assertions and axioms about the universe. The tool comprises CycL—a higher-order logic specification language—as well as a theorem proving engine for this language. The knowledge base is easily extensible [2]. Cyc can also be extended with modules that perform common tasks required by particular domains, for instance to efficiently convert between units of measure. The reasoning engine automatically determines when to invoke such modules in order to answer a query.

From our primary investigations, while not quite granting our wish for comprehending English, Cyc shows real potential to play the role for the common sense knowledge base and theorem prover that we wished for in the previous section. Its end users then will not be humans, but computer programs trying to make sense of one another.

Let us redo the above example, but this time instead of using our imagination we will use CycL to specify the meanings and invoke queries. To translate English specifications into CycL, we follow the following conventions.

CycL variables appear as underscored names. We would like to make assertions about an object and the meaning of any of its properties inside specifications. Therefore, there is implicitly a CycL variable of the same name that stands for the actual program variable. We take advantage of the fact that Cyc already knows about the `ProgramVariable` concept, and has a binary relation `variableHasName(`$ProgramVariable$`,`$CharacterString$`)` to associate each property with a name. We also added to the ontology a few binary relations: Assuming the object (`this`) and any of its properties (in our running example: `now`) are simply `ProgramVariable`s, we define a relation `variableHasProperty(`$ProgramVariable$`,` $ProgramVariable$`)` to list the properties belonging to each object. Finally through the `variableRepresents(` $ProgramVariable$`,`$Thing$`)` relation we can assert that a variable is representing something related to the real world.

6.1.1 Object and API Discovery

We will do two versions of the clock example. In the first version, the consumers need to discover the component and the API to access the time information but can assume the

[2] Cyc is available for free to researchers and academics: http://www.cyc.com/platform/researchcyc.

data will be presented in the epoch form. The `clock` component with its CycL annotations is shown below.

```
1 clock = {
2   now: /* same as before... */,
3   spec: function() {
4     return ["(variableRepresents this Clock)",
5       "(variableRepresents now
6          (SubcollectionOfWithRelationToFn
7           Integer secondsSince1970ToDate Now))"]
8   }
9 }
```

The programming environment implicitly asserts the statements `(isa this ProgramVariable)`,`(isa now ProgramVariable)`,`(variableHasProperty this now)`, and `(variableHasName now "now")` to save the user from stating the obvious. So in line 4 the user adds the fact that `this` represents a clock, a concept known to Cyc as `Clock`.

The concept of the current time is represented in Cyc by `Now`, but there is no existing individual concept of the epoch time (although one can add it). There is, however, a binary relation `secondsSince1970ToDate`($Integer, Date$) representing this concept. A fact `secondsSince1970ToDate`($num, date$) holds when $date$ is the date that is num seconds since the start of 1970. The statement `(SubcollectionOfWithRelationToFn Integer secondsSince1970ToDate Now)` denotes the (singleton) subset of integers *i* such that `secondsSince1970ToDate`(i, `Now`) holds. Thus lines 5-7 precisely state the meaning of the `now` property, i.e., it represents the current time in terms of Unix epoch time.

Time to turn to the consumers of this data. Below we have translated the previous English specifications into CycL.

```
1 var clock = K.find(
2   "(and
3     (variableRepresents that thing)
4     (typeBehaviorCapable-DeviceUsed thing
5       (MeasuringFn Time-Quantity)))")
6 if (clock) {
7   var epochMethodName = K.ask(clock,
8     "(and
9       (variableHasProperty that prop)
10      (variableRepresents prop
11        (SubcollectionOfWithRelationToFn
12          Integer secondsSince1970ToDate Now))
13      (variableHasName prop methodName)
14     )".methodName
15  var epochMethod = clock[epochMethodName]
16  var epoch = epochMethod()
17  setAngleBasedOnOnEpoch(epoch)
18 }
```

To discover the `clock` object, lines 1-5 ask the environment for an object which "*represents a device that is capable of measuring time*." (We will discuss the challenge of constructing a sentence in CycL later.) Note here the flexibility brought about by the ability to incorporate semantic decision making from within the programming language and as part of the logic of the program: there is no reliance on the name of the desired object, or even on the more general concept of a clock. To discover the correct API to access the desired data, lines 7-14 ask for the name of the property (i.e., binding for the `methodName` logical variable in the result of the query), belonging to the received object, which is supposed to represent the epoch time.

6.1.2 Data Adaptation

Now onto the second version of the consumers' code, in which they make no assumption in what format the time data is received. They simply ask the system to present them the data in the desired units. Here is the alternate version of the specifications for the `clock` object:

```
clock = {
  now: /* same as before... */,
  spec: function() {
    return ["(variableRepresents this Clock)",
            "(secondsSince1970ToDate " +
              this.now() + " Now)"
  }
}
```

Since the evaluated specification fully conveys the value of current time, here the customer of this data—e.g., the *seconds* clock hand—can simply ask the question according to the discovered object "*What is the time now in terms of seconds on a wall clock?*" Below we show the CycL translation of this query in line 4.

```
1 var clock = /* same as before... */
2 if (clock) {
3   var secs = K.ask(clock,
4     "(SecondFn sec Now)").sec
5   setAngleBasedOnSec(secs)
6 }
```

The Binary functional relation `SecondFn`($secs, date$) extracts the seconds $secs$ from a date $date$. Since for the *seconds* clock hand the seconds is the data of interest, this function was used. Therefore, evaluating the `ask` call above would invoke the following sample query on Cyc:

```
((variableRepresents that Clock) ∧
 (secondsSince1970ToDate 1394916307 Now))
                  ⟹
          (SecondFn sec Now)
```

Note that the mention of `this` in the specs of the provider of data has been replaced with `that`, since the query is from the point of view of the consumer. Also observe in the query above that the term `Now` seems to be simply playing the role of a logical variable. Yet this cannot be, since specifications from the data provider (antecedent) and the consumer (consequent) are independently written and may not be sharing free variables. Rather, the logical connection is made by both parties referring to a single shared semantic concept of `Now`, which represents the current date.

The query returns a model whose binding for the `sec` variable has the intended data (e.g., `{sec: 7}`). Thus given the current time value in terms of epoch, the theorem prover was able to convert the data in terms of wall clock seconds

and hand it to the consumer here. Note that we never had to know about or use the `now` property belonging to the provided data, since the semantic specifications stored in its `spec` property already conveys all the information (i.e., the value of current time) needed by the consumer.

6.2 Implementation

To experiment, we designed and implemented a publish-subscribe based programming language called $Navā$. The language integrates Cyc to enable the *call by meaning* programming model as we described in the previous sections. $Navā$ runs inside the browser. The language as well as a demonstration video of the running example of this paper are publicly available:

`http://www.hesam.us/nava`

$Navā$ adds distributed programming to *KScript*—a functional reactive programming (FRP) language on top of JavaScript (see [22] and also [18]). Each object is in its own separate process [3] and no state is shared between objects. We also integrated a publish-subscribe mechanism into the language to experiment with the automatic component discovery problem. Unlike the common topic-term-based subscriptions, in $Navā$ users can state CycL queries to specify subscriptions.

To avoid burdening the reader with learning the syntax of a new language, we only introduce $Navā$'s syntax and the implementation of our evaluation benchmark in this language in the appendix. It suffices to say the CycL annotations work just as described for the JS examples above. Furthermore, the code is more readable and compact because distribution, reactivity, and publish-subscribe are inherent parts of the language and there is no need for the user of the system to implement his own platform to support a dynamic component repository and discovery.

When run on a Macbook Air laptop, our queries from the clock microexample run fast enough to have the time data provider publish its data every second and see the clock GUI *seconds* hand reposition itself every second. This is despite the fact that each consumer of data invokes multiple queries that discover the data component, learn how to access the intended data, and how to adapt the values. This indicates the possibility that using a common sense language and reasoner as powerful as Cyc as part of the system runtime could in fact be practical in certain scenarios.

7. Semantic Web Service Challenge

The Semantic Web Service Challenge (SWSC) [23] has become a de facto benchmark for investigators both in academia and industry who aim to improve the state of art in the area of automated discovery and mediation of web services [20]. The goal of this challenge, according to the website, is to "develop a common understanding of various technologies intended to facilitate the automation of mediation, choreography and discovery for Web Services using semantic annotations."

SWSC presents three scenarios: (1) shipping discovery scenario, (2) hardware purchasing scenario, and (3) logistic management scenario, each involving some aspect of discovery and negotiation among various services. In each scenario, there is an English description of a requested service from the client, along with the characteristics of a set of available service providers. The challenge is to design a representation and mediation solution that given this information in each scenario can determine the set of services that can indeed satisfy a request. The first year report [23] documents the result of solutions submitted by several academic and industrial groups, each with varying degrees of success.

The goal of the SWSC closely matches that of this paper, although the scope of its three scenarios is somewhat limited compared to our setting. Unlike in our setting, there the discovery and mediation is done among a small, fixed set of similarly defined services (e.g., several shipping companies with slightly varied sets of parameters such as shipping rates, etc.), and with occasional anomalies in their descriptions (e.g., rates available only on-demand).

Nevertheless, because the benchmark is widely considered the standard and still offers a realistic representation of many challenges and subtleties for the discovery problem, we decided to evaluate our Cyc-empowered prototype by implementing the SWSC scenarios. The goal is to demonstrate how much the task is eased and the solution made more robust by giving the developer of the software components the ability to invoke real world, common sense reasoning within the programming language, whether *offline* as a development aid, or even *online* as the actual part of a program's logic.

7.1 Methodology

It was not our goal to exactly follow the SWSC guidelines since we are not submitting our solution to this challenge; the actual competition was held a few years ago. Therefore, we did not extend the capabilities of our prototype in order to provide every aspect of the benchmark. For instance, a request in these scenarios may include heuristic preferences (e.g., least price), in which case the mediation service needs to rank the matching results accordingly. The ability to discriminate between multiple matches is essential. Yet $Navā$ doesn't yet include support for this and we did not implement it. We also did not perform parts of the challenge dealing with combining services (i.e., suggesting different vendors for different products for a multiple product order). In our setting we invoke one query per object in the repository of available components and have not considered cases where multiple components are asked for within a single query. Similarly, we did not perform the task of generating a

[3] Communication can be set up locally using the JavaScript *WebWorkers* framework or over the network using *WebSockets*. See:
`http://www.w3schools.com/html/html5_webworkers.asp`
`http://www.w3.org/TR/2011/WD-websockets-20110419`

justification for negative matches, i.e., why a particular service cannot satisfy a given request. While generating justifications for positive and negative matches is quite useful, at this point we have only focused on the automation of the essential parts of the publish-subscribe system.

Below we discuss our experiment with the first two SWSC scenarios. We skip the third scenario in favor of saving time and space; although larger in size, it is not substantially different from the first scenario in terms of logical and knowledge representation complexity.

Shipping Discovery Scenario The task in the first scenario is to find suitable shipping companies that would deliver the client's package, given the specified constraints from both the customer and the service providers (e.g., destination, delivery time, price).

Hardware Purchasing Scenario In the second scenario, the product lines of several computer hardware stores are described along with technical specifications (e.g., memory, disk space, CPU speed, dimensions) of laptops and computer accessories. Given a customer's order and specification requests, the mediation solution must recommend the stores that offer a matching product.

7.2 Advantages

We include a partial listing on meaning specifications for the providers and consumers in our benchmarks in the appendix Secs. A.2 and A.3. For brevity, here we simply make a few points on the experiments.

Leveraging and Building on Existing Ontology

Generally, to write specifications in a particular domain, new knowledge (i.e., concepts and rules) has to be added to Cyc. In our shipping scenario, for example, Cyc does not know about the concept of "*latest pick up time to be considered for next day delivery*" and its ramifications. Therefore, we defined a new binary relation `latestTimeTypeForPickUp` to express this characteristic of a shipping company and wrote deductive rules to express its consequences. The key thing to note, however, is that we build on an already existing vast common sense knowledge base, and basic things need not be explained. Let us expand on this a bit.

Related to the above sample concept and the general scenario, we can leverage the existing semantic understanding in Cyc of the concepts of "an organization," "FedEx," "temporal statements," "duration," "day," "times of day," "human," "act of delivery," "geography," "currencies," "arithmetic," "units," and many more. Here are a few examples of leveraging existing knowledge.

- Once the relation `latestTimeTypeForPickUp(`$Organization$`,` $TimeOfDayType$`)` is defined, we can express any time of the day as the latest pickup time for an organization, e.g., `(TimeOfDayFn TimeOfDay-1PM (MinuteOfHourFn 35))` to denote 1:35pm or `TimeOfDay-MidnightHour` for midnight. We can rely on the theorem prover's knowledge about times of day and understanding of which one of two given times comes earlier or later.
- We can leverage Cyc's knowledge on geography. For instance, some of the shipping companies only ship to specific countries with different shipping rates, yet they charge the same flat pickup fee for any delivery. We defined a relation `pickupFeeForOrganization(`$Organization$`,`$GeopoliticalEntityOrRegion$`,` $MonetaryValue$`)` to denote this fee. However, there was no need to list out every single country served by the company, since the pickup fee is the same. We simply asserted `(pickupFeeForOrganization this PlanetEarth (Dollar-UnitedStates 12.50))`. Cyc can itself deduce, since any destination country is part of `PlanetEarth`, that this fee applies to any delivery.

Leveraging Knowledge of Units, Conversion and Arithmetic Modules

We mentioned that Cyc's theorem prover has the ability to recognize when it can avoid search by reasoning which and when efficient modules can be called upon, as part of its reasoning on a given query. For instance, it can efficiently perform arithmetic, string manipulations, and unit conversion functions when necessary. Here are a couple of examples.

- It is nearly impossible for the programmer to account for all possible discrepancies in units of data. For example, in our shipping scenario, the company may have expressed its maximum allowed package weight as `(Pound-UnitOfMass 50)`, yet the customer describes its package having a weight of `(Kilogram 30)`, or even a peculiar one `(Grain-UnitOfMass 70000)`. Nevertheless, the theorem prover is able to infer the right conversion functions to invoke in order to do its reasoning.
- We can easily include arithmetic statements as part of specifications. For example, the final fee for a delivery is stated as `(evaluate totalFee (PlusFn flatFee (PlusFn deliveryFee (TimesFn mass rate))))`. Of course these values will not be bare numbers, but rather quantities with proper units that Cyc's calculations will respect.

7.3 Implementation

Implementing the two scenarios in $Navā$ required very little programming. The bulk of the effort consisted of writing the CycL specifications; the publish-subscribe system and theorem proving backend then seamlessly use these specifications to solve the tasks. In each scenario, we instantiated one $Navā$ object for each service and one for each request. The service objects (i.e., the shipping companies and the computer products in the two scenarios, respectively) have no state other than the `spec` property holding their descriptions in CycL. The request (customer) objects each have a single line of code, subscribing in terms of a CycL query to services matching their needs.

Because $Nav\bar{a}$ is a reactive system, in case a new publisher comes along or an existing one changes its advertisements, its data gets (re)published and thus subscribers get up-to-date matches.

7.4 Results

Table 1 reports the results from the two scenarios of SWSC. The first three rows state the number of new constants, N-ary relations, and deductive rules (or *axioms*) that needed to be added to the Cyc knowledge base for each scenario, in order to be able to fully describe the services and queries. The fourth row (titled "avg service #sentences") indicates the average (among services) number of CycL sentences (i.e., a single fact over a relation) it takes to describe the service.

The remaining rows show for each query involved in one of the two scenarios the size of the query itself ("query #sentences") and the average time it took on a Macbook Air to run each within a $Nav\bar{a}$ program ("avg query time"). We obtained the average time for each query Q by measuring the time it took to get an answer from the theorem prover for the question "$S \implies Q$" for each service specification S.

In Cyc the theorem prover may be unable to arrive at an answer in time to a given query, so timeouts can occur. Also it generally takes longer for the theorem prover to prove a query unsatisfiable than to produce a model for a satisfiable query. For this reason we also report running times for each case separately in the table.

It is not of importance here to explain the queries *A1-E3*. We refer you to the challenge's report [23] for the full details. We simply make a point that the amount of knowledge that was necessary to introduce to Cyc was small. It took one of the authors roughly a day to build each scenario. The query times are reasonable (13 seconds on average). These indicate a promising possibility for practical applications of this approach in various software engineering scenarios.

Note that the second part of the second scenario (*C1-C4*) is about combining services, which as we noted was outside the scope of our tool. However, for completeness, we manually constructed CycL queries representing such combination requests to report on the query sizes and the performance of the prover.

Although the query times may be acceptable in many applications, we do see cases of long wait times in the table and a number of timeouts in the case of false queries. However, it should be noted that we have not put any effort in optimizing our querying strategy.

8. The Challenges Ahead

By no means do we claim to have a solution to the grand vision of the Semantic Web. In this section we enumerate some of the many unresolved challenges that we must address.

8.1 Working with a Meanings Lingua Franca

Any knowledge base attempting to cover a sizable portion of human common knowledge is necessarily enormous and complex. Thus writing specifications over an ontology such as Cyc can be an overwhelming task. Even in our little clock example we saw the meanings we wanted to express quickly becoming challenging to construct in CycL. To make this simple demo, as novice users we had to spend a lot of time exploring the ontology browser to find the CycL equivalents of various concepts and the constructions to build whole sentences from those concepts. Knowledge base search for appropriate terms is a known challenge with such a large knowledge base and it has a steep learning curve.

Nevertheless, there are two positive aspects about the Cyc tool specifically that can help us here. First, the Cyc system does include natural language processing capabilities, so both service advertisements and queries can be potentially written in a free-form natural language rather than CycL. At this stage, however, these features are only experimental and unavailable to the public, so we have not been able to investigate leveraging them.

Secondly, Cyc's powerful inference engine can be turned back on its own ontology to search for terms. For example, to find a function that returns a date and takes a duration at some argument position, we can run the query:

```
(and (isa f EvaluatableFunction)
     (argIsa f n Time-Quantity)
     (resultIsa f Date))
```

which results in this set:

DateAfterFn, DateBeforePreciseFn, DateAfterPreciseFn, DateBeforeFn, DateAfterDurationStartPreciseFn

Note that our vision is independent of the choice of the lingua franca for meaning specifications. The choice of Cyc for our investigations was simply speculative, and it is possible that alternative choices such as ConceptNet [4] will be more appropriate in some settings.

8.2 Specifying Behavior

The focus of this work is discovery by comparing formal specifications of advertised data and request queries. Yet this form of discovery is not necessarily the appropriate choice in all contexts. For instance, it might be more appropriate to specify the desired functionality (say to sort a list of integers) as a set of input-output pairs.

8.3 Trust, Safety, and Security

The semantic interoperability model we envisioned is too naive. The system fully trusts the specifications claimed by the providers, an entirely false assumption when operating at the scale of the Internet. Mechanisms to verify the genuineness of specifications and their adherence to the actual content are essential to study. This model also fully trusts the entire knowledge base; a single faulty rule (a piece of knowledge) entered by the knowledge base editors can cause the theorem prover to make incorrect deductions.

[4] http://conceptnet5.media.mit.edu/

Table 1. Data from implementation of SWSC scenarios I & II in $Nav\bar{a}$/Cyc on a MacBook Air 1.8 GHz Intel Core i5, 4 GB RAM. Timeout (*t/o*) set at 60 sec.

	I. Shipping Discovery				II. Hardware Purchasing			
#new concepts	3				41			
#new relations	20				7			
#new axioms	14				64			
avg service #sentences	44				3			
		avg query time				avg query time		
query	**query #sentences**	**true** (sec.)	**false** (sec.)	**all** (sec.)	**query #sentences**	**true** (sec.)	**false** (sec.)	**all** (sec.)
A1	11	0.3	1.1	0.5	19	0.3	4.0	3.8
A2	11	0.2	0.5	0.5	20	0.6	8.1	7.2
B1	11	0.5	0.7	0.6	20	0.7	7.2	6.2
B2	11	0.4	1.4	1.0	19	0.8	11	9.6
C1	13	20	57 *(1 t/o)*	35	24	1.2	–	1.2
C2	13	31	42 *(2 t/o)*	38	18	1.5	–	1.5
C3	13	5.4	47 *(3 t/o)*	38	28	1.8	–	1.8
C4	–	–	–	–	28	1.8	–	1.8
D1	12	26	14	16	–	–	–	–
E1	12	3.3	55 *(1 t/o)*	44	–	–	–	–
E2	12	27	12	18	–	–	–	–
E3	14	3.1	55 *(3 t/o)*	44	–	–	–	–
avg	12	11	26	21	22	1.1	7.6	4.1

8.4 Deficiencies in the Knowledge Base, Failures of the Theorem Prover, Lack of "Fail-Softness"

The fate of each query from the system is also severely dependent on what rules are present in or absent from the knowledge base. For instance, had the knowledge "*a timepiece is capable of telling time*" or "*a clock is a timepiece*" been missing from Cyc's base KB, the discovery in the clock example would have failed. Similarly, the case of the theorem prover automatically converting a date format from Unix epoch to clock seconds would not have worked if relevant date conversion knowledge wasn't present in the KB. More generally, the fate of every decision is put in the hands of a very complex ontology and theorem proving machinery. This is a major concern since we are proposing to utilize this common sense reasoning machinery as a black-box system.

For the envisioned programming model to be robust, further investigation is needed on how to handle the case of theorem provers failing to provide an answer. Our simple assumption of getting an `undefined` or `false` return value from a `find` or `ask` call may not be acceptable in many situations.

Lieberman *et al.* make a useful distinction between systems that use common sense reasoning in a "question-answering" form vs. "interface agents." (see [13]). Our vision in its *online* form as given falls in the first category, where the system demands and wholly depends on an answer from the reasoning engine. This severe dependency on a common sense system which is huge, complex, and unpredictable is a very real drawback. In the second category, the reasoner is only employed to make useful suggestions when possible, but its contributions are not critical in any way. This corresponds to the *offline* version of our vision. The authors of [13] suggest that using a reasoner in such a "fail-soft" manner is likely to be a more realistic usage for common sense reasoning.

8.5 Inefficiencies of General Theorem Proving

One drawback of our experimental approach, as in other logic-based solutions and compared with domain-specific solutions, is inefficiency. Invoking theorem provers can be prohibitively slow.

8.6 Discrimination in Matchmaking

The task of discrimination in matchmaking was a part of the Semantic Web Service Challenge that we did not incorporate. In the prototype $Nav\bar{a}$, the subscriber receives all published matches, clearly infeasible at the scale of the Internet. A couple of mechanisms are essential to have.

First, a subscriber must be able to provide preferences and heuristic metrics to receive only the top matches. Second, it seems necessary to have a mechanism to bind to a particular match and prohibit further matching. For example, if our clock GUI objects are already receiving data from one provider of current time, they should be able to update their subscription to stop being bothered by other matches, unless the current provider becomes unavailable.

8.7 Scalability of the Publish-Subscribe Model

It remains to be seen to what extent maintaining a repository of queryable components can be scaled up to large networks, whether examined in its offline or online conceptions. Our experiments so far have all been limited to a small number of local threads representing individual components.

9. Related Work

This section compares our work to previous works in programming languages, as well as existing solutions to the vision of the Semantic Web.

9.1 Programming Languages

The offline version of our vision is similar to a component discovery IDE feature in the Zones project [2]. Users can an-

notate components (small Scratch [26] programs) with simple natural language annotations that describe them, as well as search for components by providing a simple query. The reverse direction is also featured, that is, the system can analyze the code and suggest a possible annotation, by finding programs in the repository which have similar code and have been given annotations. However, due to being ambitious the approach is very restricted. The annotations and queries are limited to simple sentences, and the semantic connection between them is made by extracting terms within them and measuring their relatedness by consulting a backend ontology such as ConceptNet.

As far as we know, no general programming language has been put forth which integrates common sense ontology and reasoning as part of its dynamic execution model, for the purposes of discovery, data compatibility, or in general. The idea of integrating theorem provers and constraint solvers into a programming language has been visited many times in the past (see for example [11, 14, 27]), but such decision procedures operate entirely over abstract symbols. They have no embedded common sense knowledge about the reality, which is the central idea behind the vision of this paper.

There exists a budding programming language called the Wolfram Language [1], which comes with built-in algorithms and incorporates hooks into real-world data. E.g., one can easily query the list of countries in Asia, call a function to obtain the population of each country, and plot the result. Although, Wolfram is similar to our work in embedding real-world knowledge, we differ in fundamental ways.

First, to the best of our knowledge, Wolfram integrates real-world data without semantic understanding of the meanings. In this sense, Wolfram provides an instance of what is sometimes referred to as *information-rich programming*: when a programming language enables hooks to load real-world data from the cloud. Another prominent example is F#'s notion of *type providers* [28]. But it's one thing to be able to list the countries in a continent, and another thing to have semantic understanding of a "country" as a geopolitical concept with many ramifications. Because such semantic understanding is present in Cyc, by simply asserting a fact "C *is a country*," the system automatically infers myriad facts about C and its relationship to other entities in the world.

Second, the Wolfram Language is an attempt to be the *everything* language, where the intention is to be able to do everything within one programming environment. However, this is in conflict with our whole software landscape and the context of this paper. The world of software is far from being one ubiquitous paradigm, but rather an extremely heterogeneous and seemingly infinitely large set of incompatible solutions. We share the vision of the Semantic Web, i.e., not to replace all programming languages and systems with a universal thing, but rather to enable semantic understanding and communication among arbitrary components by providing a universal *glue* language for specifying meanings.

More restricted forms of the general discovery problem have been investigated in the form of various programming models. The goal of *dependency injection* [24] work is to remove hardcoding of dependencies and enable finding or replacing them dynamically. In Google's Guice [5], one component might say "*I need an object of this type and (optionally) with these annotations*," and the system will automatically provide (maybe instantiate) an object that matches those requirements, if there is a binding for that particular combination of types and annotations in the appropriate context. This is an instance of the discovery problem, but types are not expressive or modular enough [3] to work universally.

Dependency injection does not address the problem of *API discovery*, which also has a rich literature. For example, Jungloid [15] is a Java IDE tool for mining an API repository to find a chain of calls that can take a given input type to a given output type. A subsequent, dynamic tool called CodeHint [8] searches for objects in the calling context that match given boolean conditions. The conditions are expressed in terms of Java expressions and therefore cannot leverage common-sense knowledge, and discovery is limited to objects present in the program's heap.

There is also a body of related work on synthesis of component adaptors (e.g., [4, 9]) that address a form of the data compatibility problem. The idea here is to reason how to connect two components with slightly mismatched APIs by modifying the interfaces to account for the discrepancies. To the best of our knowledge, there hasn't been work in this line that leverages real-world, common sense ontologies. The approaches that we have seen tend to be problem-specific and unlikely to yet be useful in realizing the vision of automatic data adaptation in general.

Finally, type systems have been proposed that take into account the measurement units of numerical quantities (e.g., [10]). This line of work is one example of the use of real-world knowledge to specify important semantics of programs. Our vision can be thought of as a significant broadening of the scope of this idea.

9.2 Semantic Web Services Solutions

Countless solutions and technologies have been proposed, during the more than two decades since the Semantic Web idea conception. Summarizing all of these works is infeasible, yet there are a number of survey papers (see [6, 20]) on the topic. To put it very briefly, these technologies use some form of markup annotations to tie software services to real-world concepts and ontologies, and deploy a wide spectrum of formalisms and reasoning techniques towards achieving the goals of the Semantic Web Services Challenge. On a given problem in a specific domain, many of these approaches can offer a sound and effective solution.

In Sec. 3 we briefly discussed why we believe the full potential of the *Internet of software services* would not be

[5] https://code.google.com/p/google-guice

realized without naturally integrating semantic capabilities within the programming languages themselves. The primary difference between our experimental approach and the many previous solutions proposed towards the goals of the Semantic Web and SWSC is the question of whether software components themselves are realizing these goals, or there is a platform sitting on top who is in charge. Every solution presented to SWSC [23] is the latter case. Each describes its own particular platform and choices of semantic ontologies for describing and querying software services. For example, two of the challenge's well performing solutions [16] designed workflow models specifically for each scenario (e.g., "*Is fee published? If yes go to next step, otherwise request a quote*," etc.) which form part of the input to their semantic discovery and mediation platform. Each step in the workflow may involve invoking a software component in the domain.

It is easy to see why a programming languages (PL) approach as envisioned here would be empowering. Unlike in other solutions, the component (object) that represents the customer in our solution to SWSC is one and the same as the component which performs semantic discovery. The component can use the generality of the programming language it is implemented in to access and make use of the discovered component. Similarly the component providing the service itself generates its own specifications, which can be tightly connected to the runtime state. For example, the statements of various fees belonging to a shipping company can directly pull the values from the program variables representing those fees. On the other hand, in existing SWSC solutions the workflow is entirely out of the hands of the services and components themselves, nor are their semantic annotations connected with the programs in a real way.

We believe platform solutions and ontologies made by any particular group will never see widespread adoption and usability for the society at large. We borrow the belief from the AI community that the only truly universal language that has a chance of offering understanding at large is in fact the mundane, common sense knowledge. Thus we propose the adoption of a common sense knowledge base and reasoner, such as Cyc, to enable the semantic interaction among software components. Our initial experience with the SWSC highlighted the time and effort that can be saved by leveraging existing concepts in the specification language.

To describe services, a number of existing solutions (e.g. [17]) propose annotating services with a set of universally agreed-upon terms from a semantic ontology. We believe this is not a scalable solution to the discovery problem. Instead, in our vision, the provider composes a series of well-formed sentences (A) (by putting together universally accepted semantic terms) to describe what it is offering. The consumer writes a different series of sentences (Q) (by also combining universally accepted semantic terms) to describe what it wants. A and Q need not be syntactically or semantically equivalent at all. The broker uses its huge common sense knowledge (KB) to infer whether A logically, by common sense, *implies* Q; if it does, we have a match. This differs from agreeing on terms. In other words, it wasn't the agreement on what to say that made the connection between A and Q, but rather the agreement on the common sense rules understood and shared by all parties.

In [21] the authors use a Resource Description Framework (RDF)-based semantic language to specify components and queries. One interesting aspect of the system is that specifications are dynamic. The platform offers mechanisms to discover and execute a particular service offered by another component. Each component is associated with an abstract semantic model and the specification of each operation states how it updates the model of the component. However, unlike in our setting, the software component and its semantic annotations are still entirely unrelated entities. Furthermore, there are no indications in the article that the platform employs preexisting common sense knowledge or is backed by a powerful reasoning system.

10. Conclusions

In this paper we discussed the large shortcomings in today's programming practices to support the growing need of harnessing data and software components found on the Internet. The tasks of *discovery* and ensuring *compatibility* between components are for the most part done manually, making existing systems hard-to-manage and fragile and building new systems a real challenge. This problem is well known and solutions are proposed under the rubric of the Semantic Web (SW). Many researchers, outside of the programming languages community, have proposed a multitude of solutions, which naturally approach the problem from outside of the languages and software themselves.

We posit that significant progress can be achieved by introducing semantic reasoning at the level of the programs and general programming languages themselves, and we reformulate the grand visions of SW as a programming paradigm to strive for. We shared our initial experiments with the idea of utilizing a universal meanings *glue* language, ontology, and theorem prover to facilitate semantic, dynamic communication among components. We implemented a publish-subscribe-ready programming language called $Nav\bar{a}$ that employs a common sense ontology and reasoner called Cyc for this purpose. Using this artifact we demonstrated one successful instance of the realization of the dream of SW on a small microexample. We also demonstrated how various tasks from the scenarios in the SWSC are naturally enabled by this programming model.

It is our hope that the paper can encourage the programming community to engage in SW research and be an indicator that the great challenges ahead are worth undertaking.

Acknowledgments

We thank Alan Kay for the encouragement to explore Cyc towards programming.

A. Appendix

For the interested reader, here we unveil our language $Navā$ (Sec. A.1) and include partial listings of implementations of our benchmarks in CycL, the specification language incorporated in $Navā$ (Sec. A.2 and Sec. A.3).

A.1 Intro to $Navā$

We designed $Navā$ as a functional reactive programming (FRP) language, because of its well-defined notion of state and state change. Each object consists of a set of variables called *streams* in the FRP terminology. Each stream is defined in terms of a functional expression over other streams or any events. The set of current values from all streams for each object constitute its *state*. Every time an external event comes in (e.g. a timer tick or, mouse click, or in our publish-subscribe setting receiving subscribed data from the network) the current values of all streams (transitively) functionally dependent on that event are reevaluated. On quiescence the state of object is said to transit to a new pseudo time.

The charm of FRP is that state changes are well-defined and dependencies are automatically maintained, much like cells in a spreadsheet. In this setting, it is clear when the object should publish its data in a publish-subscribe system: precisely at the point of quiescence. To simplify programming, every object in $Navā$ implicitly publishes its current stream values at that time in the form of a JSON object [6]. Thus, there is no need for explicit publishing of data as would be necessary in the JavaScript version of the clock example above.

Let us revisit our micro-example once more.

Publishing

The following shows the definition of the time source object in $Navā$.

```
t = timerE(1000)
now = Math.floor(
  new Date().getTime() / 1000) on t
```

Because there is no explicit publishing, we use a timer event to make the object publish its current time data on every timer tick. The stream `t` will hold the timer value that fires every 1000 ms. The stream `now` isn't functionally dependent on `t`, but it is explicitly made so using the `on t` directive. As a result every time the timer signals, `t` gets a new value and thus `now` gets the returned value from calling the JavaScript expression `(Math.floor(new Date().getTime() / 1000))` which gets the current system time in the Unix epoch format. At this time the object implicitly publishes the current values of its streams in the JSON form.

Subscribing

$Navā$ adopts the following syntax for a subscription:

`on (`$formalName$`) (`$dataPattern$`)` $onReceiveFnBody$

So on the subscriber end of the clock example, we have the following definition.

```
clock = on (that) (that.now !== undefined) {
  return that }
epoch = clock.now
angle = setAngleBasedOnEpoch(epoch)
```

In the subscription, the expression `(o.now !== undefined)` is the $dataPattern$ as a query to state if the data matches the subscription and $onReceive$ simply returns the received object so it is stored in the stream `clock`. Since stream `epoch` is functionally dependent on `clock`, and in turn `angle` is on `epoch`, every time the network forwards the clock data, these streams will get reevaluated and thus a new angle is computed and set for the clock hand GUIs.

Meaning Annotations

$Navā$ programs can be annotated with CycL meaning specifications, just as we demonstrated earlier in the JavaScript version. Below we augment the time source object with the meanings, using the optional `: "`$\langle some\ spec \rangle$`"` annotation.

```
this : "(variableRepresents this Clock)"
t = timerE(1000)
now : "(variableRepresents now
          (SubcollectionOfWithRelationToFn
            Integer secondsSince1970ToDate Now))"
  = Math.floor(
      new Date().getTime() / 1000) on t
```

As before, if an object contains some meaning annotation it will store those as the `spec` property within its published JSON object. Note that when specifications include embedded JavaScript expressions, they will be evaluated and replaced by their values before publishing of the object occurs.

A.2 SWSC Shipping Discovering Scenario

Figure 2 shows, in CycL syntax, a sample problem-specific relation we added to the Cyc ontology related to the shipping scenario (relations are still considered *constants* in CycL terminology). Note that we are building on the already existing knowledge about the concepts of ternary relations, an organization, geographic regions, and money amounts.

Figure 3 shows a sample deductive rule we added to Cyc to be able to reason about the intended shipping queries. Note that some of the relations are problem-specific and added by us, while others (e.g., `geographicallySubsumes`) are existing knowledge.

Figures 4 and 5 list the full descriptions of a customer delivery request and shipping company, respectively. In our $Navā$ program, when an object representing a shipping

[6] We may allow the user to limit the set of streams published for optimization or security.

company with a specification like in Figure 5 comes online, the environment will ask the backend theorem prover if such a specification implies a request by a subscriber like the one in Figure 4.

A.3 SWSC Hardware Purchasing Scenario

Figure 6 shows a sample problem-specific constant we added to the ontology related to the hardware purchasing scenario. Instead of making assertions about a general laptop product to specify its various specifications, we decided to define specific product lines in the knowledge base representing a particular choice of options. In Figure 6, we are leveraging Cyc's understanding of computer hardware devices and Macbook computers to define the specific product line of "13 inch MacBooks with a 1.83GHz Intel Core Duo processor and 512 MB DDR2 RAM and a 60 GB hard drive." Furthermore, we make assertions about its technical specs, e.g., that it contains one 60 GB hard drive.

Figure 7 shows a sample problem-specific relation we added to the ontology related to the hardware purchasing scenario. In this case we use `sellsProductTypeWithId` to be able to make a statement that certain store carries items from a product line tagged with a specific identifier.

Figure 8 shows a sample deductive rule we added to Cyc ontology to be able to reason about the intended product queries. The axiom makes a general statement that "*a computer's disk capacity can be calculated by the number and the capacity of individual disk drives inside it.*"

Figure 9 lists the full description of a laptop product sold by a computer store, while Figure 10 shows a full product request from a consumer.

```
constant: shippingFeeForRegion.

isa: TernaryPredicate.
arg1Isa: Organization.
arg2Isa: GeopoliticalEntityOrRegion.
arg3Isa: MonetaryValue.

comment: "(shippingFeeForRegion ORG REGION
MONEY) means that MONEY is the flat rate
that shipping company ORG adds to any other
shipping charges for region REGION.".
```

Figure 2. A sample relation defined in CycL for the shipping scenario

```
(implies
  (and
    (shippingFeeForRegion org region deliveryFee)
    (weightForShippingPurposes package mass)
    (countryOfAddress address countryOfAddress)
    (geographicallySubsumes region countryOfAddress)
    (pickupFeeForOrganization org region flatFee)
    (orgHasShippingRateForRegion org region rate)
    (evaluate totalFee
       (PlusFn flatFee (PlusFn deliveryFee
           (TimesFn mass rate))))
  (totalShippingChargeToAddress
      org package address totalFee))
```

Figure 3. A sample deductive rule added to Cyc related to the shipping scenario. Terms added to Cyc base KB are in italicized green.

```
(actsInCapacity-MainFunction that performedBy
Shipping) // Is it capable of shipping packages?

(massOfObject package (Pound-UnitOfMass 1)) //
Package is 1 lb.
(lengthOfObject package (Inch 7)) // Package
length is 7 inches.
(widthOfObject package (Inch 6)) // Package
width is 6 inches.
(heightOfObject package (Inch 4)) // Package
height is 6 inches.
(packagePickupRequestTime package
(DateFromStringFn "2014-03-16 07:30:00")) //
Package order is placed at...
(isa address ContactLocation) // Package is
addressed to...
(fullAddressText address "105, Avenue de
la Libert. Tunis, 10002, Tunisia") // this
address...
(countryOfAddress address Tunisia) // Package
is going to Tunisia.
(cityOfAddress address CityOfTunisTunisia) //
Package is going to city of Tunis.
(canDeliverPackageTo that package address)
// Can it deliver my package to my recepient
address?
(deliveryWithinDurationAvailable that package
(DaysDuration 3)) // Can it deliver it within 3
days?
(thereExists charge (and
(totalShippingChargeToAddress that package
address charge)
(greaterThan (Dollar-UnitedStates 50) charge)))
// Will it charge less than $50 for this package?
```

Figure 4. Full description of a customer delivery request

```
(variableRepresents this ShippingOrganization) // This is a shipping organization.
(orderToPickupMinimum this (HoursDuration 2)) // Order must be given minimum 2 hrs before pickup.
(deadlineForNextDayDelivery this TimeOfDay-6PM) // Order must be picked up before 6pm for next day delivery.
(latestTimeTypeForPickUp this TimeOfDay-8PM) // Latest time pickup can occur is 8pm.
(pickupFeeForOrganization this PlanetEarth (Dollar-UnitedStates 12.50)) // Pickup fee is $12.50.
(orgHasShippingRateForRegion this ContinentOfEurope (DollarsPerPound 6.75)) // $6.75/lb fee for Europe.
(shippingFeeForRegion this ContinentOfEurope (Dollar-UnitedStates 53.5)) // $53.5 flat fee for Europe.
(orgHasShippingRateForRegion this ContinentOfAsia (DollarsPerPound 7.15)) // $7.15/lb fee for Asia.
(shippingFeeForRegion this ContinentOfAsia (Dollar-UnitedStates 60)) // $60 flat fee for Asia.
(orgHasShippingRateForRegion this ContinentOfAfrica (DollarsPerPound 10.15)) // $10.15/lb fee for Africa.
(shippingFeeForRegion this ContinentOfAfrica (Dollar-UnitedStates 75)) // $75 flat fee for Africa.
(orgHasShippingRateForRegion this ContinentOfAustralia (DollarsPerPound 7.15)) // $7.16/lb fee for Oceania.
(shippingFeeForRegion this ContinentOfAustralia (Dollar-UnitedStates 60)) // $60 flat fee for Oceania.
(orgHasShippingRateForRegion this ContinentOfNorthAmerica (DollarsPerPound 4)) // $4/lb fee for North America.
(shippingFeeForRegion this ContinentOfNorthAmerica (Dollar-UnitedStates 40)) // $40 flat fee for North America.
(orgHasShippingRateForRegion this ContinentOfSouthAmerica (DollarsPerPound 4)) // $4/lb fee for South America.
(shippingFeeForRegion this ContinentOfSouthAmerica (Dollar-UnitedStates 40)) // $40 flat fee for South America.
(orgDeliversToRegion this ContinentOfAfrica) // We deliver to Africa.
(orgDeliversToRegion this ContinentOfNorthAmerica) // We deliver to North America.
(orgDeliversToRegion this ContinentOfSouthAmerica) // We deliver to South America.
(orgDeliversToRegion this ContinentOfEurope) // We deliver to Europe.
(orgDeliversToRegion this ContinentOfAsia) // We deliver to Asia.
(orgDeliversToRegion this ContinentOfAustralia) // We deliver to Australia.
(maxWeightForShippingPurposes this (Pound-UnitOfMass 50)) // Packages must be less that 50lb.
(deadlineForDeliveryWithin this (DaysDuration 3) TimeOfDay-5PM) // Can deliver within 3 days if ordered by 5pm.
```

Figure 5. Full description of a shipping company

```
constant:  MacBook-13InchScreen-1830MHzIntelCoreDuo-512MBDDR2-HDD60GB-White.

isa: FirstOrderCollection
isa: ComputerHardwareDeviceType.
genls: MacBook-13InchScreen.

comment: "The collection of 13 inch MacBooks with a 1.83GHz Intel Core Duo processor,
  512 MB DDR2 RAM and a 60 GB hard drive.".

productTypeHasComponent: ComputerMemoryCard-DDR2-SO-DIMM-512MB.
productTypeHasComponent: IntelCoreDuo-1830MHz-ComputerProcessor.
productTypeHasComponent: Computer-HardDrive-60GB.
```

Figure 6. A sample constant defined in CycL for the hardware purchasing scenario

```
constant: sellsProductTypeWithId.

isa: TernaryPredicate.
arg1Isa: IntelligentAgent.
arg2Isa: FirstOrderCollection.
arg3Isa: CharacterString.
arg2Genl: TemporallyExistingThing.

comment: "(sellsProductTypeWithId VENDOR, PROD, STRING_ID) means vendor VENDOR sells product PROD with a
product id (as a string name) of STRING_ID.".
```

Figure 7. A sample relation defined in CycL for the hardware purchasing scenario

```
(implies
  (and
    (genls product Computer-Generic)
    (numberOfSpecifiedComponentTypeInSystemType product device count)
    (genls device HardDiskDrive)
    (relationAllInstance diskCapacity device capacity)
    (evaluate total (TimesFn capacity count)))
  (computerTypeHasLocalHDStorage product total))
```

Figure 8. A sample deductive rule added to Cyc related to the hardware purchasing scenario. Terms added to Cyc base KB are in italicized green.

```
(variableRepresents this BestBuyStore) // This is a BestBuy store.
(sellsProductTypeWithId this MacBook-13InchScreen-1830MHzIntelCoreDuo-512MBDDR2-
HDD60GB-White "00000001") // We sell white, 13 inch Macbooks with 1.83GHz CoreDuo CPU, 512MB RAM, 60 GB
hard disk with a product id of ''00000001''.
(sellsProductTypeWithIdAtPrice this "00000001" (Dollar-UnitedStates 1099.00)) // ...the price is $1099.00.
```

Figure 9. Full description of a hardware store's laptop product

```
(isa that (ProductProviderFn-Seller ComputerHardwareMarketCategory)) // Does it sell computer stuff?
(sellsProductTypeWithId that product productId) // ...and sell a product...
(sellsProductTypeWithIdAtPrice that productId price) // ...at certain price...
(productTypeHasComponent product proc) // ...which has a processor component...
(genls proc IntelCoreDuoProcessor) // ...that is an Intel CoreDuo...
(computerTypeHasProcessorSpeed product speed) // ...which has the speed...
(greaterThanOrEqualTo speed (GigaHertz 2)) // ...of at least 2 GHz...
(genls product WhiteColor) // ...which is white...
(productTypeHasComponent product memory) // ...and which has another component...
(genls memory ComputerMemoryBoard) // ...which is a RAM...
(storageCapacity memory ram) // ...which has an amount...
(greaterThanOrEqualTo ram (Gigabyte 1)) // ...which is more than 1GB...
(computerTypeHasLocalHDStorage product storage) // ...which has storage...
(greaterThanOrEqualTo storage (Gigabyte 100)) // ...of at least 100 GB...
(deviceTypeHasScreenSize product screen) // ...which has a screen size...
(numericallyEquals screen (Inch 13)) // ...of 13 inch...
(genls product MacBook-Computer) // ...and it is a MacBook...
(lessThanOrEqualTo price (Dollar-UnitedStates 1500)) // ...which costs at most $1500.
```

Figure 10. Full description of a customer product request

References

[1] The Wolfram Language. https://www.wolfram.com/language.

[2] K. C. Arnold and H. Lieberman. Managing ambiguity in programming by finding unambiguous examples. In *OOPSLA '10*.

[3] G. Bracha. Types are anti-modular. http://gbracha.blogspot.com/2011/06/types-are-anti-modular.html.

[4] C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 2008.

[5] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08/ETAPS'08*.

[6] H. Dong, F. K. Hussain, and E. Chang. Semantic web service matchmakers: state of the art andchallenges. *Concurrency and Computation: Practice and Experience*, 25(7):961–988, 2013. ISSN 1532-0634.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. 2003.

[8] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *ICSE '14*.

[9] P. Inverardi and M. Tivoli. Automatic synthesis of modular connectors via composition of protocol mediation patterns. In *ICSE '13*.

[10] A. Kennedy. Types for units-of-measure: Theory and practice. In *CEFP'09*.

[11] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL '12*.

[12] D. B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Commun. ACM*, 38(11):33–38, Nov. 1995. ISSN 0001-0782.

[13] H. Lieberman, H. Liu, P. Singh, and B. Barry. Beating common sense into interactive applications. *AI Magazine*, 25:63–76, 2004.

[14] G. Lopez, B. Freeman-Benson, and A. Borning. Implementing constraint imperative programming languages: the kaleidoscope'93 virtual machine. In *OOPSLA '94*.

[15] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *PLDI '05*.

[16] T. Margaria, C. Winkler, C. Kubczak, B. Steffen, M. Brambilla, S. Ceri, D. Cerizza, E. D. Valle, F. M. Facca, and C. Tziviskou. The sws mediator with webml/webratio and jabc/jeti: A comparison. In *ICEIS '07*.

[17] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2), Mar. 2001.

[18] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *OOPSLA '09*.

[19] M. Minsky. Commonsense-based interfaces. *Commun. ACM*, 43(8):66–73, Aug. 2000. ISSN 0001-0782.

[20] L. D. Ngan and R. Kanagasabai. Semantic Web service discovery: State-of-the-art and research challenges. *Personal Ubiqu. Comput.*, 2013.

[21] S. Nikitin, A. Katasonov, and V. Y. Terziyan. Ontonuts: Reusable semantic components for multi-agent systems. ICAS '09, 2009.

[22] Y. Ohshima, A. Lunzer, B. Freudenberg, and T. Kaehler. KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. In *Onward! '13*.

[23] C. Petrie, T. Margaria, H. Lausen, and M. Zaremba. *Semantic Web Services Challenge: Results from the First Year*. 2008.

[24] D. R. Prasanna. *Dependency Injection*. Manning Publications Co., 1st edition, 2009.

[25] D. Ramachandran, P. Reagan, and K. Goolsbey. First-orderized ResearchCyc: Expressivity and efficiency in a common-sense ontology. In *Papers from the AAAI Workshop on Contexts and Ontologies*, 2005.

[26] M. Resnick, Y. Kafai, J. Maloney, N. Rusk, L. Burd, and B. Silverman. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. Technical report, 2003.

[27] H. Samimi, E. D. Aung, and T. Millstein. Falling back on executable specifications. In *ECOOP '10*.

[28] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *DDFP '13*.