



## ONTOLOGIES AND ONTOLOGY LANGUAGES FOR WEB SERVICE DESCRIPTION

**Abstract.** Inspired from studies in ethnomethodology<sup>1</sup> and foundation of computation there are some ongoing efforts of reassessing the role which formality plays in the field of software development. As there is still no complete supporting theory about a possible alternative, we describe some observations and reasons why we think that “*the world is as opposite of formal as it is possible to imagine*” [1] and that abstractions are actually transient, shifting and negotiated. We have good reasons to doubt that the current mechanism of layers of effective formality is the right mechanism, and we believe that it is important to raise some interest in the idea of introducing “*real-world sloppiness*” as part of the programmatic effectiveness. Taking such radical approach would have some profound consequences on web service description and automation. As part of this paper we'll make an attempt to insinuate some of these consequences.

### I Roadmap

The roadmap for this paper is a bit odd. First because in it's more substantial part it's following the structure of a talk as it was given by Gregor Kiczales at OOPSLA 2007, and second because the notions we present are so new, that they sound almost as a pseudo-science. The second is the main reason we dedicate the next six sections on introducing some foundational ideas and observations. We start with a brief description on how programming languages are conflicting with things like design patterns and how mathematics are in conflict with things like how people work. In support to these conflicts we give two examples of how useful informality is in our field, and to enhance the contrast we discuss our definition for a program as an *effective formal abstraction of computation*. Since this discussion is around languages we present the idea of *indexicality of language* as we know it in our natural language. Then we move on to a study about *intentionality* and why it is a foundational idea for *computation*. As part of this study we see what's the connection between *language indexicals* and the process of *registration*, but we don't explain in much detail why we consider the world *fundamentally sloppy*, relying on the assumption that the reader can already sense this by intuition. The sixth of these first six sections combines these ideas into a radical thesis. The last two sections are an attempt to insinuate some consequences these notions can have on web service ontology design and on the ontology design as a whole. Our consequences section is a bit lengthy, but we compensate with a brief summary at the end.

### 2 Two conflicts

We'll start by introducing *two conflicts* that have bothered and confused programmers for years. Gregor Kiczales talked about these conflicts at (OOPSLA, 2007):

---

<sup>1</sup> <http://en.wikipedia.org/wiki/Ethnomethodology>

The first conflict is between programming languages at one hand and things like design patterns and Mylyn<sup>2</sup> on the other hand, and that conflict is between the kind of power that languages give you versus the flexibility patterns have when dealing with high-level structures and situation particulars.

The second conflict has to do with foundations - it is about how we should conceptualize software and software development. An the conflict there is between the very mathematical formal foundation that we currently use and things like the way people work, make sense of the world and intentionality. [2]

Both these conflict have to do with reassessing the role that formality plays in our field. We believe that we need to incorporate formality with some of the degree of sloppiness and imprecision the world around us is characterized with.

### 3 Samples of effective informality

Gregor Kiczales uses Mylyn as an example for how a definition of a *task*<sup>3</sup> can be defined without being formalized, how it can evolve organically from watching developers, how adaptive it can be when it is not rigidly enforced and how it is socially constructed and negotiated. The main point about Mylyn is that it successfully introduced an approach for light-weight management of higher-level structures.

Another sample Gregor Kiczales gives is about a mid 90-ties experiment with an attempt to formalize design patterns - what the authors<sup>4</sup> had discovered was that the generated code was often difficult to integrate with the existing code and that surprisingly the most useful aspect of the tool turned out to be decidedly low-tech: the book text itself in HTML form with hyperlinks and cross-references. The point in this example is to show that the power of design patterns is in their flexibility to be used in surprising cases, combinations and integrating them with situation particulars. Formalizing the patterns takes that power away as no longer the meaning, applicability and form were socially mediated.

So these are two good examples we can use for showing the power of informality. Let's take a look at programming languages on the other hand!

### 4 Programming languages

Programming languages are all about formality. Gregor Kiczales defines programs in his talk as *effective formal abstractions of computation*. Further on he elaborates on the meaning of each of the words - effective, formal and abstraction. We find important to share his understanding of these terms.

---

<sup>2</sup> <http://www.eclipse.org/mylyn/>

<sup>3</sup> or concern

<sup>4</sup> (Chambers, Harrison, Vlissides, POPL 2000)

By *effective* we mean not only useful, but the particular thing that programs do the physical work of generating a computation. Editing a program will eventually result in generating a different computation. In this meaning design patterns are not effective, but programming languages are.

By *formal* we mean simply that programs are formal systems containing formal symbols that we can manipulate, but we also mean that these systems are crisp and discrete and in which the ontology they start with is very clear and well-defined.

And by *abstraction* we mean that the programs are abstractions of computation - the programs don't need to include in them all of the computation particulars.

The effectiveness of programming languages is extraordinarily powerful. Furthermore effectiveness comes with the ability to compose pieces of computation into bigger pieces of computation. And large part of a programming language is designed to control this composition. In order to make compositions to work we focus on having units that are well-defined, context insensitive, semantics free, orthogonal, etc - the whole effort is to be able to come up with notions that are clear enough to find and compose and reason with a small set of principles about what will happen. Again *abstraction* comes to help, but with a different function - when controlling compositions abstraction is used as minimization of effectiveness - one can not effectively change what is hidden from him.

In the current state of affairs we build computer systems with design approaches like layering, encapsulation, exposed APIs, using frameworks or building domain specific languages - all used with the goal to define interfaces that present effective abstractions. And the result is that we end up with effective formality all the way down to the machine code. When you're building a digitality, you are building a formality - at some point it is true that programs must be crisp, because the programs are engineered, but we must not forget that:

**1. Programs are engineered by people in social settings.**

**2. Computations are emerged socially.**

Gregor Kiczales refers mostly to the ethnomethodological studies of (Suchman, 2009), where she discusses a lot about how people work together; how we construct shared understanding and what's the role played by artifacts and to the research summarized in (Smith, 1998) where he focuses on the conceptual foundations of computations and information.

The perceived common threads of both works is that the *world isn't formal or it is at least not formal all the way down*. And that this has real consequences for software.

## 5 Indexicality of language

Ethnomethodological studies shift the focus of how we react in respond to the world towards the notions that our everyday social practices render the concepts we share about the world:

3. The objectivity of situations of our action is achieved, rather than given.

4. A central resource to achieving objectivity of situation is language, which stands in a generally indexical relationship to the circumstances,...
5. As a consequence of the indexicality of language, mutual intelligibility is achieved on each occasion of interaction with the reference to situation particulars rather than being discharged once and for all by a stable body of shared meaning. [3]

We use a lot of *indexicals* in our natural language. Indexicals are adverbs like *here* and *now*, but indexicals are also all expressions that rely on their situations for significance. And with the help of indexicals we don't need too many words to precisely establish what is that we intend and what is that the other party understands. The shared understanding is not precise and formal, but yet it is completely adequate for the purposes of the particular interaction.

We believe that programming languages can work that way too - at least to a certain extent.

## 6 Intentionality and foundation of computation

(Smith, 1996) started as a project in foundation of computation, but it became exploration of *intentionality and ontology*, because of the author's belief that the only way to explain computation is in terms of intentionality.

The question that interests us is *how intentionality works and arises*. But before that we must give some idea of what is intentionality?

*Intentionality is the power of mind to be about, to represent, or to stand for things, properties and states of affairs. The puzzles of intentionality lie at the interface between the philosophy of mind and the philosophy of language.*<sup>5</sup>

But intentionality is really the ability to have *aboutness* relationships.

This definition matters to us because when it comes to develop software we have to handle many kinds of artifacts - some of them abstract, some of them concrete, some of them formal and some of them not. We also have to handle all diverse properties and relationships among these artifacts or between the people producing the software, or between both of them - some of these relationships are effective and formal, some of them are not. And on top of this we have to deal with numerous theories and areas of work that focus on parts of this software development ecosystem - development tools, denotational semantics, computational semantics, real-world semantics, product-line architectures, requirements analysis, etc. All of these relationships are intentional and therefore the intentionality steps up as a spanning theory to talk about the entire software development ecosystem - from people to running code.

There are some objections to this theory. Some are trying to distinguish between original intentionality and derived intentionality - only people can possess original intentionality, computation can at its best have only derived intentionality. Other objection is that since computation is founded on the ground of formal symbol systems, it is semantic free and it can't have an about

---

<sup>5</sup> Stanford Encyclopedia of Philosophy

relationships, so it can't be intentional. Please note, that first our efforts are targeting the developers, which unarguably possess intentionality, and second that we don't really need our artifacts to have intentionality, but to know what the theory of intentionality has to say about them.

The key idea of (Smith, 1998) is:

The world is fundamentally characterized by an underlying flex or slop - a kind of slack or "play" that allows some bits to move about or adjust without much influencing, and without being much influenced by other bits. [1]

So eventually semantics are really about things that exceed your effective grasp. Notice that this is completely opposite to programming languages where the semantics are completely in your effective grasp and where at least the ontology is fixed. But here in this slop you don't need to do that - the idea of precise individuation criteria just isn't right.

The other key idea in (Smith, 1998) is about the process of *registration*, which explains how we're still able to individuate things and still have clear thoughts no matter that at the bottom things are sloppy and zesty. By *register* he means things like parse, make sense of, find there to be, structure, take as being a certain way. Registration, similar to indexicals is something very context sensitive.

## 7 Combining the themes of ethnomethodology and intentionality

After combining these themes Gregor Kiczales makes an attempt to formulate a radical thesis in which he states that in order to work effectively with higher-level issues the formality is not the foundational idea and the layers of effective formality is not the right mechanism. Effectiveness has to be more sloppy, allowing it to be negotiated, periodic, partial, evolving and build around something like registration.

## 8 How can these notions be applied to web services?

Web services extend the Web from a distributed source of information to a distributed source of service. Semantic Web has added a machine-interpretable information to the Web content in order to provide intelligent access to heterogeneous and distributed information. In a similar way, Semantic Web concepts are used to define intelligent web services, i.e. services supporting automatic discovery, composition, invocation and interoperation. [4]

The ultimate goal is to be able to perform tasks as *to book a flight from Innsbruck to Madrid, for next Wednesday and with a fixed maximum price*. [4] We are not saying that the current research in the field of semantic web services is irrelevant to solving this problem, we're just saying that in the sample above there are only entities that are part of fixed and well-defined ontologies - none of the terms *booking, flight ticket, departure town, destination town, date, maximum, price* needs further clarification, because these terms have a precise meaning. Any of these meanings is part of the web service ontologies that would be used in the composition. The semantics of these services are either build-in or modeled on top of the existing web service description with some sort of annotations that add capabilities for semantic interpretation<sup>6</sup>. And this is going to work as long as

---

<sup>6</sup> e.g. WSMO-Lite

we use terms and concepts that are precise and formal in nature - or also characterized by being low-level.

But let's consider the task of *finding an affordable apartment in Sofia not too far from a swimming pool and with a nice view*. What means to be *affordable* would vary widely depending on the subjects incomes. The perception of what should be considered *not to far* will be different for a car owner and for a pedestrian. And finally what is a *nice view*? Even if there are all needed services available for discovery and composition - services like *apartments for rent*, *swimming pools register* and a *mapping service*, and even if all heterogeneity and interoperability issues between these services are solved, it would still be impossible to perform the request until a shared understanding for all of the imprecise terms has been built. Currently, such understanding is embedded directly in the service ontology, but no matter whether the ontology avoids to use too specific terms in favor of more general and incomplete term, it would still be unable to deal with such kind of sloppiness.

A possible solution for this problem is to:

**1.Design the web services to be decoupled from its semantics - thus we would use externally bind-able semantics or contexts for the service.**

This first step will guaranty that we are not entirely dependent on the ontology build-in the domain specific language of the service. We would be able to either tell the service how it is supposed to interpret its semantics or even bind the service to the semantics we need.

At this moment we're only able to say that this should be similar to the aspect-oriented programming or to the meta-object protocol in Lisp, without going into speculations how it will work or look like to have a service that can execute client-specified cross-cutting concerns or pinpoints.

**2.Implement a suitable registration process to establish an adequate understanding for indexicals used in service messages.**

This second step will guaranty that the client and the service *make sense of the world or parse* in a way that both parties will understand each-other. The result of this process should be a mutual agreement on what means *affordable*, *not to far* and *nice* for that particular situation.

## 9 Consequences

At the start, the intended purpose of this paper was to take some of the existing ontologies and ontology languages for web service description and to try evaluate them to a set of engineering design criteria as described in one of the most widely cited works in ontology<sup>7</sup> (Gruber, 1993). Later on as we've begun to incorporate the ideas of *sloppiness* and to reassess the current design principles it became evident that the set of design criteria we use to evaluate ontologies should change.

---

<sup>7</sup> [http://en.wikipedia.org/wiki/Ontology\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Ontology_(computer_science))

Let's take for example the *clarity*-criteria. In the description given by (Gruber, 1993), currently this criteria is limited to produce only binary results - an ontology is either clear or unclear - no further information can be extracted. This is a conclusion we infer not only from the current state in the field of software development<sup>8</sup>, but also from our believe that Gruber's formulation of clarity fails to acknowledge the importance of social negotiations, situated actions and objectivity achieved through indexical languages, as we've discussed earlier:

An ontology should effectively communicate the intended meaning of defined terms. Definitions should be *objective*. While the motivation for defining a concept might arise from social situations or computational requirements, the definition should be independent of social or computational context. *Formalism* is a means to this end. When a definition can be stated in logical axioms, it should be. Where possible, a *complete* definition (a predicate defined by necessary and sufficient conditions) is preferred over a partial definition (defined by only necessary or sufficient conditions). All definitions should be documented with natural language. [5]

Let's try to analyze the above description. It starts by simply saying that if we make modifications to the terms in an ontology, the intended meaning should also change accordingly - this refers to *effectiveness* in the very same sense as of calculation effectiveness. Changing the meaning of terms should directly change the calculation process. As we've seen, such effectiveness is currently attained through layers of formality. One can get the feeling that formality has also been used in the sense of objectivity, but further Gruber elaborates on the purpose of formality as a means to decouple definitions from any social and computational context. Admitting that *all definitions should be documented with natural language* contradicts in a way with the whole description. It shows that Gruber on a subconscious level or with his intuition could sense that the formality all the way down may not be the right mechanism. In support to that Gruber also writes about the *partial definitions*, which are also formal definitions but with partially defined *necessary or sufficient conditions*. Partial definitions can somewhat resemble the concept of *sloppiness* as they usually give some freedom for interpretation, but in the world of formality there is no way to establish a negotiated understanding between the ontology server and the ontology consumer, except for when there are humans involved and natural language is in use. Partial definitions are used for situations where it is more practical to decouple the ontology from too specific concepts.

We won't go any further in analyzing the rest of the criteria given by (Gruber, 1993). At this point the reader should have some idea about the direction in which these thoughts are leading to, so here we can formulate some of our conclusive suggestions.

## **I. The architecture of the semantic web<sup>9</sup> as a layered system of formal ontologies<sup>10</sup> all the way down is *impractical*.**

---

<sup>8</sup> software development as producing formality all the way down

<sup>9</sup> and the semantic web services as a subset of the web

<sup>10</sup> based on industry standards and notations

Like most software engineers we're always seeking the most effective way to get our job done - we're all driven by pragmatism. But right now formality is bit in the way of effectiveness. We can point one main disadvantage that we're not sure if it's obvious. One can start feeling it only when the systems he builds or uses grow to the size of the systems we use today. And this disadvantage has emerged directly from the way in which we've been using formality.

One very common usage is when we design a system as a multi-layered structure. One way of layering is by defining higher-level concepts only by using already defined lower-level concepts, thus constructing a ontology. Ontologies cease to be practical the moment when in order to understand a concept you need to get acquainted with a substantial part of the ontology's elements. This is exactly what we ran into when we've tried to assess the currently developed ontologies and ontology languages<sup>11</sup> for web service description - infinite stacks of incomprehensible layers, each layer introducing it's own languages and paradigms. The mistake we've made through the years is to try scale vertically with this approach.

Another way of using the layering technique is by insulating components into layers, thus components in one layer are limited to communicate only with each other and only with the components from the nearest surrounding layers. The goal of this approach is to make possible when studying a complex system to limit your degree or interest only to a certain layer. Yet again such systems cease to be practical the moment when in order to understand well the components in a layer you're forced to explore more and more layers to the extend of being forced to get to know the whole system.

In conclusion, we must say that scaling a system with layers on top of previous layers should be done with a sense of deprecation. We admit that currently bottom-up modeling looks like the only feasible approach when it comes to extend the meaning of already running systems. An alternative approach is to embed a mechanism which allows effective renegotiation of meanings, thus allowing all parties to extend a system without wrapping it with additional layers.

## **2. The current design of the semantic web can only be effective to humans and AIs which posses *intentionality*.**

We are aware that ontologies are processed with automation tools, but by introducing the concepts of sloppiness, social negotiations, situated actions and objectivity achieved through indexical languages we want to shift the focus to the productivity if those who create the tools and the ontologies itself. No matter how semantic the web gets, the rigidness of formality we use today can posses only the intentionality of it's creators. In it's current architecture the semantic web will not exceed the scope of a conventional expert system.

In support to this insinuation, we would like to bring to your attention the design decisions Stephen Wolfram did for the Wolfram Alpha<sup>12</sup> computational search engine:

But what about all the actual knowledge that we as humans have accumulated?

---

<sup>11</sup> e.g. SAWSDL and WSMO-Lite

<sup>12</sup> <http://www.wolframalpha.com/>

A lot of it is now on the web—in billions of pages of text. And with search engines, we can very efficiently search for specific terms and phrases in that text.

But we can't compute from that. And in effect, we can only answer questions that have been literally asked before. We can look things up, but we can't figure anything new out.

So how can we deal with that? Well, some people have thought the way forward must be to somehow automatically understand the natural language that exists on the web. Perhaps getting the web semantically tagged to make that easier.

But armed with *Mathematica* and NKS I realized there's another way: explicitly implement methods and models, as algorithms, and explicitly curate all data so that it is immediately computable.

It's not easy to do this. Every different kind of method and model—and data—has its own special features and character. But with a mixture of *Mathematica* and NKS automation, and a lot of human experts, I'm happy to say that we've gotten a very long way. [6]

With his deep insight and perceptiveness Stephen Wolfram was able to sense that no matter how tagged the web becomes, there is still other way and in this way, at least for the moment, it's not possible to proceed without explicitly modeling and curating data and without using human experts.

By incorporating our radical ideas we can start experimenting with agents which communicate with each other with a form of an indexical language.

Imagine web services that are able to renegotiate their contract and QoS parameters depending on the specific needs of each consumer, instead of just being linearly traversed and selected by some rigidly automated agent. We are not limiting the possibilities to only unidirectional interaction between parties. Communication between agents<sup>13</sup> should resemble the objectivity achieved in social situations through the use of indexical languages.

### **3. Ontology and ontology language design should focus on providing a degree of sloppiness instead of just focusing on crispness and coherence.**

There are two major problems we're facing as software engineers - first is the process of understanding requirements and transiting the extracted knowledge to a working program, and second is coping with the ongoing changes. While formality is good for its properties of being effective at easily changing the calculation process and for its ability to provide effective *compositions*, it is also infamous with the difficulties when tracking side effects of changes and when trying to find the requirements corresponding to a fragment of the program. Requirements are encompassing not only some achieved knowledge about the intention and form of the software product, but also the situation, the plan and the social perspective on the software development

---

<sup>13</sup> computer-computer or human-computer

process. In the current state of affairs a program is rarely self-exploratory<sup>14</sup> and the only way of tracking the effect of change is by thorough testing or by using mathematical proofs<sup>15</sup>. But even when the desired behavior is assured, the software engineers are still incapable of tracking all the *why's* and *what's* for the design decisions made for a fragment of code. This is an important knowledge which may affect their judgement and the evolution of the program.

In the course of the evolution of a program contradicting definitions and statements can occur and it is almost impossible to maintain full coherence. Discrepancies are usually detected when the user or the tester runs into defects. But even when all noticeable defects are removed and ontology's coherence is ensured, it is pretty hard to verify if the program will reflect the current intent of the stakeholders.

The knowledge about our intent is rarely crisp and it is manifested in situated actions which are in indexical relationship with the *circumstances*. Programmer's decisions may be taken not solely based on best practices and established engineering principles, but could also be affected from the circumstances in the social setting where the actual programming happens - the programmer may decide on whether the allocated time or priority of the task requires the implementation to comply with some principles or best practices, or he can ignore them.

There is a high degree of sloppiness in the whole software process. Programming languages are not designed to reflect this sloppiness and their lack of flexibility presents a barrier when it comes to cope with change, but non-programming tools like Mylyn are doing it in a way. The next step is to try incorporating sloppiness and the technique of registration into the design of ontologies and in the programming languages. Instead of performing modifications in terms of rigidly changing the programming code, the modification should resemble more a negotiation process. This should add an unparalleled degree of flexibility and completely new levels of effectiveness when a change is needed.

## 10 Summary

The following is a list of consequences or concluding suggestions we've made an attempt to formulate:

1. The architecture of the semantic web as a layered system of formal ontologies all the way down is *impractical*.
2. The current design of the semantic web can only be effective to humans and AIs which possess *intentionality*.
3. Ontology and ontology language design should focus on providing a degree of *sloppiness* instead of just focusing on *crispness* and *coherence*.

Specifically we've defined a couple of design approaches that could be used with semantic web services in order to fulfill a more ambitious goal than the goal the current semantic web is pursuing:

---

<sup>14</sup> for further information on self-exploratory programs, please check (Alan Kay, 2006)

<sup>15</sup> used in the formal software processes

1. Design the web services to be decoupled from its semantics - thus we would use externally bind-able semantics or contexts for the service.
2. Implement a suitable *registration* process to establish an adequate understanding for *indexicals* used in service messages.

These suggestions were deduced from the facts that:

1. Programs are engineered by people in social settings.
2. Computations are emerged socially.

## References

1. Brian Cantwell Smith, On the Origin of Objects, The MIT Press, 1998
2. Gregor Kiczales, Effectiveness Sans Formality (and foundations of software development), 2007 OOPSLA Keynote talk, slides available at <http://www.cs.ubc.ca/~gregor/papers/kiczales-oopsla-07-for-viewing.pdf>, audio available as podcast at [http://www.oopsla.org/podcasts/Keynote\\_GregorKiczales.mp3](http://www.oopsla.org/podcasts/Keynote_GregorKiczales.mp3)
3. Lucy Suchman, Human-Machine Reconfigurations: Plans and Situated Actions (Learning in Doing: Social, Cognitive and Computational Perspectives), Cambridge University Press, 2nd edition, 2006
4. Ruben Lara, Holger Lauser, Sinuhe Arroyo, Jos de Bruijn, Dieter Fensel, Semantic Web Services - description, requirements and current technologies, Innsbruck University
5. Thomas R. Gruber, Toward Principles for the Design of Ontologies Used for Knowledge Sharing, 1993, available at <http://tomgruber.org/writing/onto-design.pdf>
6. Steven Wolfram, Wolfram|Alpha is Coming!, Wolfram Blog, available at <http://blog.wolfram.com/2009/03/05/wolframalpha-is-coming/>
7. Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumatra, Andreas Raab, Steps Toward the Reinvention of Programming (a Compact and Practical Model of Personal Computing as a Self-Exploratorium), 2006, available at <http://irbseminars.intel-research.net/AlanKayNSF.pdf>