Journal of Universal Computer Science, vol. 21, no. 11 (2015), 1405-1424
submitted: 10/5/14, accepted: 12/1/15, appeared: 1/11/15 © *J.UCS*

SKYWare: The Unavoidable Convergence of Software towards Runnable Knowledge

Iaakov Exman
(Department of Software Engineering,
Jerusalem College of Engineering, Jerusalem, Israel
iaakov@jce.ac.il)

Juan Llorens, Anabel Fraga, Jose María Alvarez-Rodríguez
(Department of Informatics,
Carlos III University of Madrid, Leganés, Spain
{llorens, afraga, jmalvarez}@kr.inf.uc3m.es)

Abstract: There has been a growing awareness of deep relations between software and knowledge. Software, from an efficiency oriented way to program computing machines, gradually converged to human oriented Runnable Knowledge. Apparently this has happened unintentionally, but knowledge is not incidental to software. The basic thesis: runnable knowledge is the essence of abstract software. A knowledge distillation procedure is offered as a constructive feasibility proof of the thesis. A formal basis is given for these notions. Runnable knowledge is substantiated in the association of semantic structural models (like ontologies) with formal behavioral models (like UML statecharts). Meaning functions are defined for ontologies in terms of concept densities. Examples are provided to concretely clarify the meaning and implications of knowledge runnability. The paper concludes with the runnable knowledge convergence point: SKYWare, a new term designating the domain in which content meaning is completely independent of any underlying machine.

Keywords: SKYWare, Runnable Knowledge, content, meaning functions, densities, knowledge distillation
Categories: D.2.13, D.2.0, D.1.0

1 The Brief Fortuitous History: Convergence of Software and Knowledge

Software was born as a very technical discipline to program a general purpose computing machine, and its main concern was efficiency. By assimilating concepts such as abstraction, encapsulation, persistence, ownership, economical value, etc., it gradually moved away from the machines that it was supposed to serve, starting to be relevant to other non-computing oriented stakeholders. Similarities and relations to Knowledge became striking. Since its inception, after the Second World War, Software shifted from a linguistic-syntactic-imperative viewpoint containing variables and simple commands, to a composite-semantic-descriptive viewpoint containing sophisticated objects and complex constructs in which semantics has a central role, as is typical of Knowledge. Something much deeper than expected has happened.
This evolution is shortly reviewed in this historical introduction, highlighting turning points of importance. The central claim of this introduction is that convergence to

knowledge has been an unconscious process only guided by pragmatic purposes – e.g. ease of development and maintainability.

1.1 Pre-History: The Emergence of Software

Software can be first recognized in association with computing machines. Such are the symbols written and read in the Turing machine's tape. Such is the set of commands put in the von Neumann computer's memory before the machine starts running, for efficiency reasons.

1.2 Classical Antiquity: The struggle towards abstraction

Programming languages enter the stage. Their proliferation caused interesting twists. Some languages intended to specialize in mathematical calculation, like FORTRAN, or to fit business oriented needs as COBOL. Others were associated with the ambitious goals of Artificial Intelligence (AI). With the strong AI hypothesis, machines began to lose importance, as intelligence could be common to machines, humans and higher animals. LISP, a functional language, gave up efficiency as a main goal, to enable a program to reason about programs themselves. The strict separation of commands from data became blurred. Logical languages, such as PROLOG, showed that it is possible to substitute commands – the imperative way – by assertions – the descriptive way. Another step, this time towards structure, was taken by escaping the chaotic-spaghetti vision of programming. With the emergence of object-oriented languages, around SIMULA, it became clear the efficiency trade-off for abstraction and a better "understanding" of software.

1.3 Middle Ages: Software Engineering and Reuse

At a certain stage of this brief history, engineers began to understand that it is a waste of efforts to rewrite software again and again with no process and control, still in terms of efficiency. The very concept of Software as an independent field of technology and engineering began to take form. But it took some time for those ideas to mature.

1.4 Renaissance: Model Transformations

The practical application of Software (SW) Reuse forced the focus on standardization and formalization. During this golden period, the SW Reuse community developed the concept of Domain Engineering, the Product Lines (PL) approach, software development with Frameworks or Generators, Components Based Software Engineering (CBSE) and the important concept of SW Asset as an economic resource. Software filled its backpack with metric formalizations, process management, quality and cost estimation. Formalization led to great advances in representation, understanding, and differentiation from hardware: diagrams with abstract representation of software laid the grounds of one of the most important steps towards Knowledge: the Unified Modeling Language (UML) and its specializations (e.g. SysML).

Recent approaches to capture reality based on model transformations, MDE (Model Driven Engineering) /MDA /MDD were a further step towards software as the way to "program" knowledge within computable machines.

1.5 Industrial Revolution: Knowledge Management and Engineering

The software history's industrial revolution was presumably attempted in Japan, when trying to understand the success of the Japanese manufacturing companies at the time. Knowledge management appeared as a discipline whose main goal is to understand, control and manage knowledge from the economic point of view, i.e. an asset with strategic value for an organization.

Knowledge Engineering, initially coupled with AI to differentiate semantic structures from data/information, was transformed with due importance paid to knowledge management and information retrieval. It evolved to support more complex information/knowledge meta- models, familiar to software engineers.

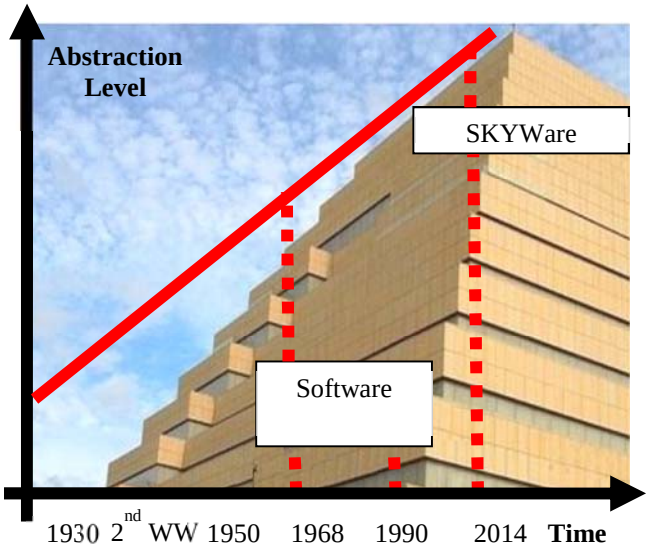

Figure 1: The Software Ziggurat – The Software Ziggurat – Software is the medium to distil knowledge from raw data. Software is itself also pure knowledge, in its upper abstraction levels. This figure shows the gradual increase of software abstraction levels along the non-linear time axis, typical of creative outbursts.

These perceptions, clarified the vision of Knowledge for the field: a semantic representation, with a clear value to a variety of stakeholders, whether humans or machines, see *Figure 1*.

1.6 Modernity: Software-Knowledge Convergence

One has innumerable overlapping cases between Software and Knowledge. Here are just a few examples:

a. Software requirements' elicitation – translation of informal requirements to software specification is a case of concept recognition problem;
b. Software model checking – an industrial application technique, uses AI's temporal logics;
c. Software (Knowledge) Representation – a venerable Software Engineering problem on the pursue to standardize Analysis and Design;
d. Software Reuse - Compose new software by domain modeling with knowledge schemas, or querying repositories by advanced IR algorithms;
e. Web-Services comparison – web-services, as distributed software system components, are compared by means of their ontologies.

The perception of Software/Knowledge similarities raised interest in the nature of these relations, and originated activities dedicated to Software and Knowledge (conferences, journals, handbooks, etc.). We think that time is ripe to substitute the "and" connective between the allegedly different fields, by a systematic treatment of their blend and relations. One finally explicitly asks the question: is this convergence of software and knowledge accidental? We believe not.

2 Background

This section provides a detailed timestamps and references to the software-knowledge historical convergence, summarized in the paper introduction.

2.1 The Emergence of Software

The software pre-history totally overlaps with computing machine pre-history, from the 1930's to the end of the Second World War. Names are so well-known – Turing and von Neumann – that there is no need of references.

2.2 The struggle towards abstraction

Programming languages proliferation lasted from the 1950's to the 1960's, to our days: cf. FORTRAN [Backus et al., 1957], COBOL [Hopper, 2012]. Artificial Intelligence (cf. the strong AI hypothesis [Penrose, 1989]) produced LISP [McCarthy, 1960], and PROLOG [Colmerauer and Roussel, 1996]. Steps towards structure started with Dijkstra's [Dijkstra, 1979]. Object-oriented languages began around SIMULA [Dahl et al., 1968].

2.3 Software Engineering and Reuse

As early (or late) as 1968, engineers understood that it is a waste of efforts to rewrite software from scratch [Mcilroy, 1969] and with no process and control [Naur and Randell, 1969]. Software engineering began to take form.

2.4 Model Transformations

During the late 1980's to the early 1990's, the SW Reuse community developed Domain Engineering [Neighbors, 1989], and the product lines (PL) approach.

Advances in representation, and differentiation from hardware include: Warnier & Orr diagrams, Jackson [Jackson, 1975]. Later on [Booch, 1994], [Rumbaugh, 1991], [Jacobson et al., 1992], [Martin, 1992] or [Harel, 1987], among others, laid down the grounds of the Unified Modeling Language (UML) [Rumbaugh et al., 1999].

Software acquired metrics' formalizations [Basili and Freburger, 1981], process management [Humphrey, 1989], quality and cost estimation [Boehm, 1981], and formal initiatives (SEMAT [54]). Recent approaches to capture reality were based on model transformations [Bézivin, 2001].

2.5 Knowledge Management and Engineering

The software industrial revolution was presumably attempted in the early 1990s in Japan [Nonaka and Takeuchi, 1995]. Knowledge Organization Systems (KOS) are well-studied [Baker et al., 2013]. Assorted models proposed [Gómez-Pérez et al., 2007] by diverse disciplines, are taxonomies, glossaries, thesauri, topic maps, ontologies, SKOS (Simple Knowledge Organization System), UML [Rumbaugh et al., 1999] and RSHP [Llorens et al., 2004]. A KOS, a Knowledge Repository [Applegate, 2007], is a collaborative system for people to retrieve organizational knowledge assets. These systems act like Knowledge Management Systems (KMS) [Denning, 2001]. They use inference to generate new information, filling document sections, and displaying why/why-not reasons.

2.6 Software-Knowledge Convergence

Overlapping of Software-Knowledge concepts, and techniques are: a) model checking [Clarke et al., 1999]; b) Information (knowledge) Representation [Maron and Kuhns, 1960] [Salton et al., 1975] [Gruber, 1993] Robertson [48], [Gómez-Pérez et al., 2007] [Antunes et al., 2007] [Fraga, 2010]; c) Software Representation [Rumbaugh et al., 1999]; d) Software Reuse [Prieto-Díaz, 1991]; e) Web-Services comparison [Paolucci et al., 2002].

2.7 Consequences and Implications

Software and Knowledge activities surge as conferences e.g. SEKE (Int. Conf. Software Engineering and Knowledge Engineering), journals e.g. IJSEKE (Int. Journal of Software Engineering and Knowledge Engineering), KAIS (Knowledge and Information Systems, an Int. Journal), Information Systems or handbooks [Haskins, 2011] to name a few. Aside a previous attempt to unify them [Llorens and Prieto-Díaz, 2003], time is ripe to delete the "and" connective – SKY (Int. Software Knowledge Workshop).

Traceability [Leffingwell, 2002], for verification and validation, is essential to search abstract software in repositories (e.g. SSBSE-Int. Symposium Search Based Software Engineering), using knowledge retrieval algorithms, e.g. by interestingness [Exman, 2009]. Taxonomic/Ontological representations are ways to cope with

scalability [Morillo et al., 2006]. Barnes and Bollinger [Barnes and Bollinger, 1991] suggested the reuse of human problem solving ability.

3 The Runnable Knowledge Thesis

From this section onwards, we switch from an historic to a more formal constructive approach. We shall argue that there are deep reasons to believe that Knowledge is the essence of Software. We start by formulating our basic thesis:

Runnable Knowledge Thesis
Any abstract software sub-system can be distilled into pure Runnable Knowledge.

This is a thesis and not a theorem as artifacts of a certain kind – knowledge elements – impart meaning to artifacts of a different kind – abstract software.
Software artifacts are syntactic constructs having a design nature. Assuming good practices of software design, implies that software has a purpose. But, software may be mal-designed; software – e.g. classes – may be even devoid of meaning (i.e. just organization units). On the other hand, knowledge artifacts are a formal expression of meaning. The natural meaning one could assign to Runnable Knowledge in the above thesis is the purpose of the software from which it was distilled.

Thus, the Runnable Knowledge Thesis is telling us that the purpose of a piece of software can be derivable from the abstract software itself (cf. traceability [Settimi et al., 2004] [Leffingwell, 2002]). We provide a proof of feasibility for this thesis in a constructive fashion. It is shown that given any software asset, there is a formal consistent procedure to distill a corresponding pure runnable knowledge. To this end, we first carefully characterize each term in the thesis.

3.1 Abstract Software

The most widespread form of software is a sub-system (see e.g. INCOSE Handook [Haskins, 2011]) written in a high-level programming language, or its respective executable code. Abstract software is obtained from software by modeling it by means of a widely accepted modeling language. Without loss of generality, we take the standard UML (Unified Modeling Language) as our defining model. A UML model is a set of directed labeled graphs.

Within UML one models structure and behavior in separate. In a structure model – typically a class diagram – nodes represent classes, and edges represent relationships among classes. These relations are generic and are restricted to a few types only (basically inheritance, composition, aggregation and association).

In a UML behavior model – we choose statecharts as the generic behavior model [21] – nodes represent states and edges stand for transitions between states. Statecharts are the behaviors assigned to software structure models. More formally,

abstract software is an ordered pair of {[class diagram],[statechart]}, which themselves are respectively an ordered quadruple and ordered pair of sets[1]:

$$\text{Abs\_Software} = [\ [\{C\},\{F\},\{V\},\{R\}]\ ,\ [\{S\},\{T\}] \quad (1)$$

where C=Classes, F=Functions, V=Variables, R=Relationships, S=States and T=Transitions.

3.2 Knowledge

Knowledge is a discrete set of concepts formally characterized by directed labeled graphs. These may take a variety of equivalent forms. Without loss of generality, we choose ontologies as a standard knowledge representation. In ontologies, nodes represent concepts and edges stand for relationships among concepts.

More in detail, the most common underlying conceptual models for ontologies representation (RDF from the W3C consortium [Hayes, 2004], or others, like our own RSHP [Llorens et al., 2004]) are based on a subject-predicate-object paradigm. This paradigm can be simplified to Concept-relationship-Concept.

Ontologies are analogous to UML structure models. One easily builds hierarchies upon both kinds of graphs. Exactly the same terms that appear as UML classes may appear as concepts (nodes) in ontologies. The important difference is that relationships within ontologies may be of any kind, thus may be much more specific than the restricted types in UML classes.

3.3 Runnable Knowledge

An entity is defined to be runnable abstractly if it has been associated with a behavioral model (a set of states and transitions – i.e. a statechart). Running in this broader sense means to make transitions among states along the time axis. Novel tools may be needed to actually run the entity, in contrast to plain executable software code.

By the previous definitions, runnable knowledge can be seen as an ontology that has been associated with a statechart.

Intuitively, since an ontology is analogous to a UML structure model, it lacks a behavior model in order to fully characterize an abstract software. The associated statechart provides the lacking behavior. More formally, runnable knowledge is an ordered quadruple of sets:

$$\text{Runnable\_Knowledge} = [\ \{C\},\{R\},\{S\},\{T\}] \quad (2)$$

where C=Concepts, R=Relationships, S=States and T=Transitions.

3.4 Distillation of Runnable Knowledge

Distillation of runnable knowledge is a selection procedure of a consistent sub-set of terms – the concepts[2] of relevance – and their respective relationships, starting from a

[1] Although Objects are basic structural constituents in UML, their omission in our definition is not considered essential as they are mainly included for snapshot situation modeling purposes.

given abstract software entity[3]. The selected concepts' sub-set may be a proper sub-set, for which the distillation is not strictly reversible[4].

3.5 Thesis Feasibility – A Constructive Procedure

Having defined the thesis' terms, we provide a constructive procedure serving as a feasibility proof. The procedure to distill runnable knowledge from abstract software is as follows:

Runnable Knowledge Distillation Procedure

1. *Ontology Concepts* – are extracted from the structural representation of software (class names, functions and/or variables, in UML);
2. *Ontology Relationships* – are obtained from UML association labels (not from the association types), directly from inheritance (e.g. *is-subtype-of*) and composition types, or inferred from class functions and/or variables.
3. *States* and *Transitions* – are directly obtained from the UML behavior model, of the respective classes;
4. *Ontology-Statechart association* – sub-statecharts are associated with ontology concepts obtaining a whole runnable knowledge piece.

The statechart is not just an arbitrarily juxtaposed behavior model for the runnable knowledge. The statechart indeed contains the runnable part of the knowledge, complementing the ontology. This procedure starts from abstract software, which is usually the case for software sub-system programs.

Any other software assets, such as specifications, user guides, tests, etc., can be converted into a universal representation, using RDF [Hayes, 2004], RSHPs (relationships) [Llorens et al., 2004], or similar, as the basic entity. Thus, any software asset may be first transformed into a UML model, with subsequent application of the distillation procedure.

The distillation procedure is seemingly simple. But it is far from trivial. For instance, since the number of UML classes and the number of ontology concepts is not necessarily one-to-one, additional considerations must be involved in the sub-set selection.

3.6 Meaning Functions

Knowledge conveys meaning. We define meaning functions, to obtain quantitative measures of knowledge, enabling precise comparisons of knowledge content.

---

[2] We take the liberty to directly assign meaningful terms, i.e. concepts, to class terms, assuming that classes were well designed.

[3] That belongs to the application domain.

[4] To allow recovery of the original abstract software one persists the complement of the selected sub-set. If the selected sub-set is the whole set, distillation is reversible.

We call *Concept Density* δ ({C}{R}) of an ontology, any monotonic increasing function of the cardinalities of the sets of concepts and relationships in the respective ontology, normalized by a measure of the sub-system size.

Definition 1 – Meaning Functions

Meaning functions μ are defined as inverse functions of the Concept Densities:

$$\mu(\{C\}\{R\}) = \frac{1}{\delta(\{C\}\{R\})}. \quad (3)$$

Thus, meaning functions are monotonic decreasing functions of the cardinalities of sets of concepts and their relationships. The smaller the sets, the greater the value of the meaning function. In other words, the lesser the graph complexity, the greater the value of the meaning function. The intuition behind the definition is that meaning increases with better understanding, which corresponds to simplicity – one needs less concepts to describe a well-understood idea.

Although the form of δ is left unspecified, we assume that well-behaved meaning functions with the desirable monotonicity characteristics are sufficient for the purposes of this paper.

An example of a function δ is the normalized product of the number of concepts by the graph density of relationships [Diestel, 2012]. A more sophisticated example of δ is a normalized function describing the time/space complexity of a knowledge computation over the ontology.

3.7 The Monotonic Distillation Theorem

The above definition is applied in the following theorem:

Monotonic Distillation Theorem
A meaning function value cannot decrease as a result of distillation of pure Runnable Knowledge out of abstract software.

A proof outline is as follows. From the definition of meaning functions they are monotonic decreasing functions of sets of concepts and relationships. Since distillation involves selected sub-sets, these are either smaller or at most equal to the original set.

The distillation procedure outcome is a meaning function that either increases or at least remains constant relative to the initial sets of concepts/relationships. Therefore, a higher level in the software hierarchy is indeed “more abstract than” a lower level.

4 Knowledge Distillation: Examples

In this section we provide concrete examples of runnable knowledge distillation, to illustrate the idea.

The first example – the Observer – is a well-established design pattern of classical software. Its software knowledge is distilled from the class diagram.

The second example – the Stadium Wave – is a much higher level abstraction. It is described from the start by an ontology, with a partial distillation.

The third example again uses the Wave to illustrate the transition to an UML diagram, needed for knowledge distillation from all kinds of software documents.

4.1 Design Pattern: Observer

The Observer is a well-known design pattern found in the GoF book [Gamma et al., 1995]. Its purpose is to establish a one-to-many dependency, such that when one subject changes state, the many observers are notified and updated automatically.

We analyze the generic class diagram in [Gamma et al., 1995] to extract the ontology concepts. It has four classes divided into two roles: subject and observer. Each role has an abstract and a concrete class. The abstract classes are concise expressions of the pattern purpose. The concrete class in each role, essentially allows implementation of specific functions of the application in which the pattern is embedded. The latter functions are irrelevant to the pattern purpose. Thus, we take only two concepts corresponding to the two roles. These are seen in the next *Figure 2*.

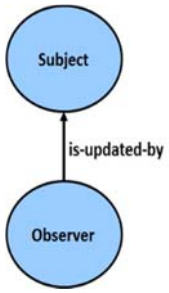

Figure 2: The Observer Design Pattern Ontology – It characterizes the design pattern by two concepts "Subject" and "Observer" linked by one relationship "is-updated-by".

Now we generate the respective statechart. The subject role is either waiting unchanged or has changed and must notify its observers. The observer role is either up-to-date or was notified of a change, and needs updating. The two roles exist and behave in parallel. Here we ignore transitions attaching/detaching observers from the subject list. Our statechart only refers to the dynamics of already attached observers.

The statechart – see *Figure 3* – is the juxtaposition of the two roles' sub-statecharts. The pair of graphs "observer ontology & observer statechart" represent together the runnable knowledge of the Observer design pattern.

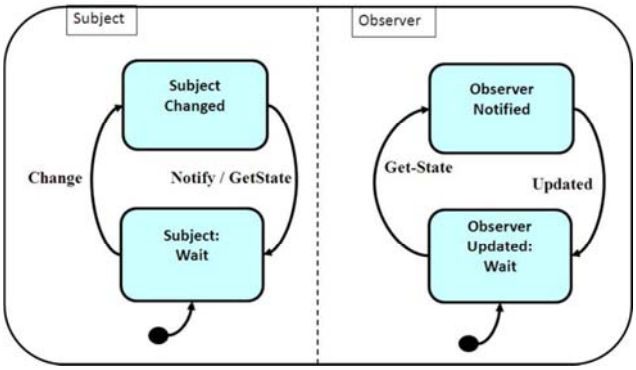

Figure 3: The Observer Design Pattern Statechart – It shows the states and transitions of the two roles "Subject" and "Observer", that exist and behave in parallel. Each role contains by itself two exclusive states.

4.2 The Stadium Wave

Imagine a stadium where people get up and sit down based on a signal given by a wave director. When successive groups of *spectators* briefly stand up and raise their arms, the result is a wave – like natural sea waves – of standing spectators traveling through the crowd, even though spectators only move vertically in their seats.

This is a highly abstract example of pure runnable software. We use standard software representations, even though the description is solely in terms of the above natural language concepts. It is indeed natural to start from the Wave ontology.

There is no need whatsoever to code it in a programming language in order to understand its runnable dynamics.

4.2.1 The Wave Ontology

The ontology has one class *person* with two sub-classes: *Spectator* (in the crowd) and *Wave director*. These sub-classes have different object properties as seen in *Figure 4*. These properties are represented as associations.

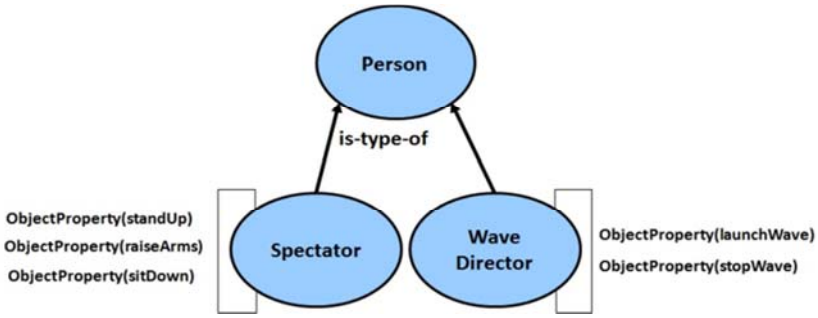

Figure 4: Wave example participants in an Ontology. It includes two sub-classes of Person and their respective Object Properties.

4.2.2 The Wave Structure: Class Diagram

The sub-classes *Wave Director* and *Spectator* (in the crowd) are seen as a class diagram in *Figure 5*. In addition to the object properties, pay attention to the move signal sent to the next person.

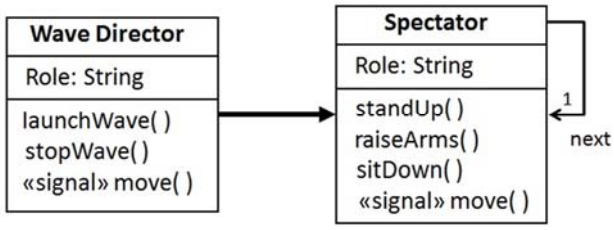

Figure 5: UML class diagram – It has two classes: Wave Director and Spectator. Besides the referred functions it shows that each spectator after sitting down sends the move( ) signal to the next spectator instance.

4.2.3 The Wave Behavior: Statechart

The statechart of the Spectator in *Figure 6* is quite simple. It has two states: seated – the default state – and standing. The pair of graphs Person Ontology and Spectator Statechart – suitably complemented by a Wave Director statechart – represent together the Runnable Knowledge of the Wave.

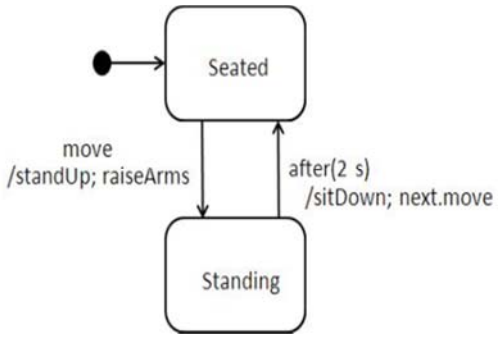

Figure 6: Statechart of the Spectator in the crowd. Such a person has two possible states: either seated or standing. The Seated state is marked as the default. While s/he stands up s/he also should raiseArms. If the person is standing it should sitDown after a short time, say after 2 seconds, and cause the next person to move.

4.3 Generic Example at RSHP representation level

Software knowledge does not consist only of UML models and programming language code. It also includes tests and all kinds of entities such as specifications and design documents at various levels.

It has been claimed that using basic knowledge conceptual models, like RDF or RSHP, one can represent in UML any kind of software entity [Llorens et al., 2004]. In this way, one can distill runnable knowledge from any entity. We now show such a concrete example using RSHPs to represent the Wave. Suppose we look at the Wikipedia article about Wave (audience) [5] as a typical *users' guide* of the Wave. There one finds the text fragment:

"The wave is … achieved in a packed stadium when successive groups of spectators briefly stand and raise their arms. Immediately upon stretching to full height, the spectator returns to the usual seated position."

One needs first to recognize and extract key concepts from the text, such as successive, spectator, stand, raise arms. To obtain the RSHP representation of successive spectators standing we take the following steps:

a. An artifact A_{text} of type text is created with a first relationship (RSHP 1):
$$A_{text} = \{ \text{(RSHP 1)} \}$$

b. Each spectator is represented as a KE (Knowledge Element):
$<KE_{Spectator}>$ = A reference to term Spectator

c. Finally, RSHP 1 keeps a relationship between two Spectator instances:
RSHP 1 = { $<KE_{Spectator}>$, <Successor>, $<KE_{Spectator}>$ }

One could similarly obtain representations of the relationships <standUp>, <raiseArms> and <sitDown>. If the source artifact is an UML diagram itself, we would create a corresponding artifact Auml of type UML. The important point is that the same RSHPs would appear whether the source is textual or UML. Thus RSHP representations of different kinds of documents are mutually traceable. In other words, whatever the kind of the source document, it can be traced to an UML representation. Any kind of software document is thereby runnable. In the next section we deal with different modes of runnability, explaining the purpose of running a text documentation or any kind of document.

5 Runnable Knowledge Modes

In this section – still referring to the Wave – we probe deeper into the meaning of abstract software as *Runnable Knowledge*, asking two questions:

a. Why is it *runnable*? Runnability is not an incidental property of software. Runnability is essential to meaning, as it increases understanding.
b. Is the *software actually abstract*? It is abstract in the sense of total independence from the underlying machine.

These two issues are argued below in three Runnable modes of Knowledge.

5.1 Thought Experiment: The Wave in the Head

The Ontology-Statechart association is the complete Runnable Knowledge for a given sub-system. For instance, for the Wave, *Figure 4* together with *Figure 6* is the

[5] http://en.wikipedia.org/wiki/Wave_(audience)

complete knowledge for a *Spectator* in the crowd. In order to better understand the *Wave* we perform a thought experiment: we run the *Spectator* in the crowd statechart (in *Figure 6*) in our head. For each spectator instance we imagine that s/he standsUp() and raisesArms(), sitsDown() and finally moves() to the next spectator. This imaginary run of the Wave is depicted in the UML Objects' diagram in *Figure 7*.

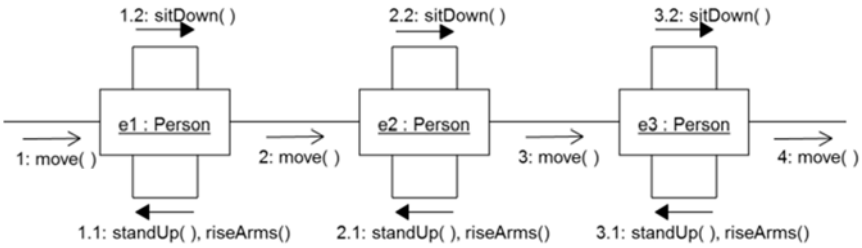

Figure 7: Wave Dynamics given by the UML Objects Diagram. One sees three instances of the Person, a Spectator in the crowd. The messages displayed for each person are: standup( ) and raiseArms( ), sitDown( ) and finally move( ) to the next person.

The thought experiment is stopped once one reaches the feeling of grasping the wave. In *Figure 7* we have stopped after 3 person instances.

Runnability as a source of software understanding is a well-known characteristic of software development. This occurs while locally debugging short pieces of a large software system in a typical IDE (Integrated Development Environment). A more recent argument is the motivation behind agile development approaches. Again, in order to locally acquire pieces of understanding, one runs small chunks of an otherwise very large and difficult to comprehend system. However mysterious, the head in which we run the thought experiment, we have not programmed it to run the Wave, and the brain is certainly not a specialized Wave runner. The Wave in the head is indeed abstract.

5.2 Watching a Wave Movie

A second Runnable Knowledge mode is – instead of running the Wave in our head – just to watch a Wave movie, say in YouTube. A very good example of the Wave director managing a Wave in a stadium is seen in the movie found in the YouTube web-site[6]. Another example of the unfolding of a Wave in a stadium is seen in the YouTube movie[7]. One clearly sees the wave moving along the spectators' crowd. After a very short time of the Wave running in the movie one catches the Wave idea. Indeed, *runnability* increases understanding. Is the Wave movie actually abstract?

6

Wave - Biggest Wave Ever:
http://www.youtube.com/watch?v=H0K2dvB-7WY

7 Rutgers Stadium Wave: http://www.youtube.com/watch?v=3NxLh-3DdaE

Although the YouTube application needs a machine to run, there is nothing in YouTube that makes it specific to any particular movie. Neither the machine, nor YouTube were developed with the Wave in mind. Engineers have clearly designed and manufactured both the machine and the YouTube application. We know for sure that the Wave is completely abstracted from the lower hierarchy levels.

5.3 Simulation of a Wave

A third Runnable Knowledge mode is a classical software simulation of a Wave. One could develop a program that simulates and displays a Wave with animated spectators in a Stadium scenery (see e.g. for a physics approach with simulations [Farkas et al., 2003]). Running the animation facilitates understanding of the Wave nature, as the reader can see by himself by watching a visual simulation[8].

Since the animation runs upon a machine, is the Wave actually abstract?

Again we are led to the same arguments as before. Good programming practices mean that we should not write an inflexible hardwired Wave code from scratch. We rather would use a generic event simulation program and embed in the generic program the abstract Wave represented by the ontology and respective statechart. Summarizing:

a. Runnability clarifies and increases understanding of the concepts and relationships appearing in the Wave ontology and its statechart.

b. The Wave is abstract since its concepts and relationships do not show up at all in any of the lower levels of the hierarchy, in any of the three Runnable Knowledge modes.

6 Discussion, Conclusions and Future Work

Beyond a formal basis and detailed techniques, SKYWare embodies the deep transformation from a machine-centered computing view into a human-centered vision.

6.1 Software as Content

If one asks people nowadays what is software?, the widely accepted response is a bag of content oriented systems – Google search, images, movies, Facebook-like social networks and smartphone applications.

The elementary units of content are concepts and their relationships, meaningful for human users. Their meaning cannot be decomposed into lower level units, neither into bits, nor into any alphabet. Content units are rather composable and now perceived as runnable.

Extrapolating from today, the SKYWare society will be populated by human-centered, visible, sociable and collaborative knowledge artifacts. A recent example of this trend is the reversal of attitude toward robots, from purely functional, opaque and mechanistic systems to robots with a human appearance and sociable behavior.

---

[8] Wave, Populating Stadium with Golaem Crowd, visual Wave simulation: http://www.youtube.com/watch?v=PG0AzQMKixc

6.2 Future Work: The MDE Tools

MDE – Model Driven Engineering – has been stated as an important goal of software and systems engineering. It is easier in terms of efforts invested to develop and debug high level models, than lower level programs.

Once one has the insight that high-level abstract software is Runnable Knowledge, one can profit from this perception in a pragmatic sense. MDE models are not just UML; they have an important knowledge component.

Software engineers will need to develop a variety of novel integrated tools to make concrete the updated MDE. For instance, development tools smoothly integrating ontology manipulation like Protégé – as seen in the screen print in *Figure 8*– with UML manipulation.

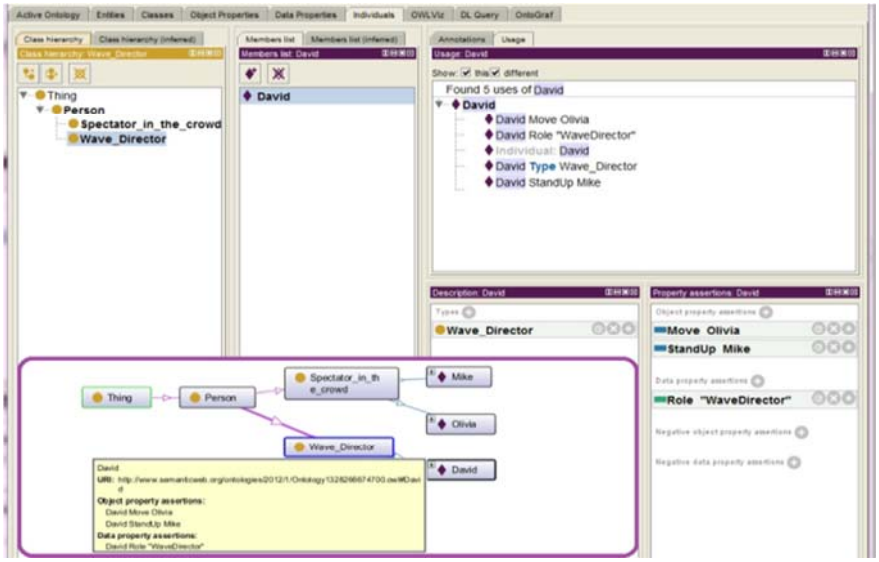

Figure 8: The wave example – from Figure 4– represented as an Ontology using the Protégé 4.1 tool. The Person concept is a sub-type of Thing. Three person instances are also depicted. David has the Wave Director Role. Olivia and Mike are Spectators (in the crowd).

We expect an evolution of current practices into a well-defined partition between knowledge software modelers, with a strong orientation to abstraction, in contrast to classical component developers and integrators. MDE implies that from the upper level Runnable Knowledge one will be able to automatically generate and regenerate software. The already observable evolution of current UML/SysML tools – to edit, generate and reverse engineer software – is expected to achieve the sophistication and reliability of compilers which translate from currently high-level languages to machine languages. Compact representations are more efficient in terms of storage and communication. It will be preferable to transmit compact runnable knowledge to other locations and generate locally the explicit lower level software.

6.3 Main Contribution: Runnable Knowledge

The main contribution of this work: the essence of abstract software is Runnable Knowledge. Runnable Knowledge is first of all a kind of knowledge. It is SKYWare, the most compact expression of the contents of a whole software system, its purpose, its components.

Figure 9: SKYWare – The full weight of the Software system is suspended from the floating SKYWare. Inspired by Magritte.

Runnable Knowledge is indeed runnable, in two clear and distinct senses. It is itself runnable in the abstract by making transitions and changing states along the run. This abstract run is visualizable for better understanding, as was exemplified by the Wave. SKYWare is also runnable after conversion to classical lower level software by current and future MDE tools. In this sense, the full weight of the whole software system – as in *Figure 9*– is carried by the compact SKYWare.

References

[Antunes et al., 2007] Antunes, B., Seco, N., Gomes, P.: “Using Ontologies for Software Development Knowledge Reuse"; Proceedings of the Artificial Intelligence 13th Portuguese Conference on Progress in Artificial Intelligence; EPIA'07; 357--368; Springer-Verlag, Berlin, Heidelberg, 2007.

[Applegate, 2007] Applegate, L. M.: Corporate Information Strategy and Management: Text and Cases; McGraw-Hill, Inc., New York, NY, USA, 2007; 7 edition.

[Backus et al., 1957] Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., Nutt, R.: “The FORTRAN automatic coding system"; Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability; IRE-AIEE-ACM '57 (Western); 188--198;ACM, New York, NY, USA, 1957.

[Baker et al., 2013] Baker, T., Bechhofer, S., Isaac, A., Miles, A., Schreiber, G., Summers, E.: “Key choices in the design of simple knowledge organization system (SKOS)"; Web Semantics: Science, Services and Agents on the World Wide Web; 20 (2013), 35--49.

[Barnes and Bollinger, 1991] Barnes, B. H., Bollinger, T. B.: “Making reuse coste effective"; IEEE Softw.; 8 (1991), 1, 13--24.

[Basili and Freburger, 1981] Basili, V. R., Freburger, K.: “Programming measurement and estimation in the software engineering laboratory"; J. Syst. Softw.; 2 (1981), 1, 47--57.

[Bézivin, 2001] Bézivin, J.: “From object composition to model transformation with the mda"; Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39); TOOLS '01; 350; IEEE Computer Society, Washington, DC, USA, 2001.

[Boehm, 1981] Boehm, B. W.: Software Engineering Economics; Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981; 1st edition.

[Booch, 1994] Booch, G.: Object-oriented Analysis and Design with Applications (2Nd Ed.); Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[Chang, 2001] Chang, S. K., ed.: Handbook of software engineering & knowledge engineering; World Scientific, River Edge, NJ, 2001.

[Clarke et al., 1999] Clarke, E. M., Jr., Grumberg, O., Peled, D. A.: Model Checking; MIT Press, Cambridge, MA, USA, 1999.

[Colmerauer and Roussel, 1996] Colmerauer, A., Roussel, P.: “History of programming languages ii"; chapter The Birth of Prolog, 331--367; ACM, New York, NY, USA, 1996.

[Dahl et al., 1968] Dahl, O.-J., Myhrhaug, B., Nygaard, K.: “Some features of the SIMULA 67 language"; Proceedings of the Second Conference on Applications of Simulations; 29{31; Winter Simulation Conference, 1968.

[Denning, 2001] Denning, S.: The Springboard: How Storytelling Ignites Action in Knowledge-Era Organizations; Butterworth-Heinemann, 2001.

[Diestel, 2012] Diestel, R.: Graph Theory, 4th Edition; volume 173 of Graduate texts in mathematics; Springer, 2012.

[Dijkstra, 1979] Dijkstra, E.: “Classics in software engineering"; chapter Go to Statement Considered Harmful”, 27--33; Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[Exman, 2009] Exman, I.: “Interestingness - a unifying paradigm - bipolar function composition"; KDIR; 196--201; 2009.

[Farkas et al., 2003] Farkas, I., Helbing, D., Vicsek, T.: “Human waves in stadiums"; Physica A: Statistical Mechanics and its Applications; 330 (2003), 1--2, 18 -- 24.

[Fraga, 2010] Fraga, A.: “A methodology for reusing any kind of knowledge at low cost: Universal knowledge reuse"; PhD Dissertation-Computer science and Engineering; Carlos III University of Madrid (2010).

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software; Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gómez-Pérez et al., 2007] Gómez-Pérez, A., Fernández-López, M., Corcho, O.: Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the

Semantic Web. (Advanced Information and Knowledge Processing); Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Gruber, 1993] Gruber, T. R.: “Model formulation as a problem-solving task: Computer-assisted engineering modeling"; International Journal of Intelligent Systems; 8 (1993), 1, 105--127.

[Harel, 1987] Harel, D.: “Statecharts: A visual formalism for complex systems"; Sci. Comput. Program.; 8 (1987), 3, 231--274.

[Haskins, 2011] Haskins, C.: “Systems Engineering Handbook. A Guide for System Life Cycle Processes and Activities"; International Council on Systems Engineering (INCOSE); INCOSE-TP-2003-002-03.2.2, 3.2.2, (2011.

[Hayes, 2004] Hayes, P.: “RDF semantics"; Technical report; World Wide Web Consortium (2004).

[Hopper, 2012] Hopper, G.M.: “The mother of COBOL”, http://cs-www.cs.yale.edu/homes/tap/Files/hopper-story.html (2012).

[Humphrey, 1989] Humphrey, W. S.: Managing the Software Process; Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[Jackson, 1975] Jackson, M. A.: Principles of Program Design; Academic Press Inc., Orlando, FL, USA, 1975.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-oriented software engineering - a use case driven approach; Addison- Wesley, 1992.

[Leffingwell, 2002] Leffingwell, D.: “The role of requirements traceability in system development"; (2002).

[Llorens and Prieto-Díaz, 2003] Llorens, J., Prieto-Díaz, R.: “Is ks=(d+i+s+k)*e + km?"; SIGSOFT Softw. Eng. Notes; 28 (2003), 2, 11.

[Llorens et al., 2004] Llorens, J., Morato, J., and Genova, G. RSHP: An information representation model based on relationships. In: Ernesto Damiani, Lakhmi C. Jain, Mauro Madravio (Eds.), Soft Computing in Software Engineering (Studies in Fuzziness and Soft Computing Series, Vol. 159), Springer (2004), pp 221-253.

[Maron and Kuhns, 1960] Maron, M. E., Kuhns, J. L.: “On relevance, probabilistic indexing and information retrieval"; J. ACM; 7 (1960), 3, 216--244.

[Martin, 1992] Martin, J.: Object-oriented analysis and design; Prentice Hall, Englewood Cli_s, N.J, 1992.

[McCarthy, 1960] McCarthy, J.: “Recursive functions of symbolic expressions and their computation by machine, part i"; Commun. ACM; 3 (1960), 4, 184--195.

[Mcilroy, 1969] Mcilroy, D.: “Mass-produced Software Components"; J. M. Buxton, P. Naur, B. Randell, eds., Proceedings of Software Engineering Concepts and Techniques; 138--155; NATO Science Committee, 1969.

[Morillo et al., 2006] Morillo, J. L., Fuentes, J. M., Prieto-Díaz, R., Astudillo, H.: “Incremental software reuse"; ICSR; 386--389; 2006.

[Naur and Randell, 1969] Naur, P., Randell, B., eds.: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO; 1969.

[Neighbors, 1989] Neighbors, J. M.: \Software reusability: Vol. 1, concepts and models"; chapter Draco: A Method for Engineering Reusable Software Systems, 295--319; ACM, New York, NY, USA, 1989.

[Nonaka and Takeuchi, 1995] Nonaka, I., Takeuchi, H.: The knowledge-creating company: How japanese companies create the dynamics of innovation; Oxford University Press, New York, 1995.

[Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T. R., Sycara, K. P.: “Semantic matching of web services capabilities"; Proceedings of the First International Semantic Web Conference on The Semantic Web; ISWC '02; 333--347; Springer-Verlag, London, UK, UK, 2002.

[Penrose, 1989] Penrose, R.: The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics; Oxford University Press, Inc., New York, NY, USA, 1989.

[Prieto-Díaz, 1991] Prieto-Díaz, R.: “Implementing faceted classification for software reuse"; Commun. ACM; 34 (1991), 5, 88--97.

[Robertson, 1977] Robertson, S.: “The probabilistic character of relevance"; Information Processing Management; 13 (1977), 4, 247--251.

[Rumbaugh, 1991] Rumbaugh, J.: Object-oriented modeling and design; Prentice Hall, Englewood Cliffs, N.J., 1991.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., Booch, G., eds.: The Unified Modeling Language Reference Manual; Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[Salton et al., 1975] Salton, G., Wong, A., Yang, C. S.: “A vector space model for automatic indexing"; Commun. ACM; 18 (1975), 11, 613--620.

[Settimi et al., 2004] Settimi, R., Cleland-Huang, J., Khadra, O. B., Mody, J., Lukasik, W., DePalma, C.: "Supporting software evolution through dynamically retrieving traces to uml artifacts"; Proceedings of the Principles of Software Evolution, 7th International Workshop; IWPSE '04; 49--54; IEEE Computer Society, Washington, DC, USA, 2004.