PhD Progress Report 3

Loránd Kedves
2017/2018. II.

Idea Projection

"What is Software Anyway?"

Doctoral School of Information Science
University of Pannonia, Veszprém, Hungary
Supervisor: Dr. Botond BERTÓK

Heading

Welcome. This was my 3rd progress report about my PhD research on Dynamic Knowledge Representation and Software Synthesis at University of Pannonia, Veszprem, Hungary. I am currently changing this long and misleading title to Idea Projection, with a provoking question at the subtitle, What is Software Anyway? If you are interested how this question can be asked seriously in such a presentation, please follow the links to start your own journey with the guidance of Bret Victor, then I recommend watching some lectures of Alan Kay.

Outline

I will summarize my research in one slide and tell you what I think software is, then I continue with a clearer definition and context. The heart of this presentation is the Idea Projection slide followed by two actual examples of using this approach. I close it with a summary of my academic progress, and a bonus slide that I had to delete from the original presentation because of the 15 minutes time limit.

What is Software?

I graduated as programmer in 1994, worked as developer, architect, system analyst and software quality assurance engineer in the past 20+ years. If there is one lesson I had to learn, that should be: all information system must have an up-to-date, detailed plan. Without that you have trouble everywhere: requirement negotiations, budget and resource management, quality issues and maintenance.

In information science we suppose to learn the definition of information: information is something you don't know and can't derive from the facts and rules you already know. Putting these statements together, there is an important conclusion: the system-specific information content of any source code should be zero: it should not contain anything that was not derived from the plans.

Then why do we write source codes? Because now this is the only way to communicate with the runtime (hardware or operating systems) and the external software modules we use (database, web server, workflow manager, message bus, …). And because they change all the time, we must write new codes to do the same task again and again, ensuring that there will be deviations, errors all the time, and will lose most lessons learned in the previous implementations.

What can we do then? Step back from the source codes. Look at the simple “Hello, world” example from different times. Which is “the real” hello world? None of them, they are the same idea serialized in different forms for different environments. “The” hello world is this graph. The knowledge that you have an application called “HelloWorld” containing a message with the greeting text, sent to a component that can display it. This applies to all information system! 1: you know it in this detail to write the code, and 2: your implementation is reverse engineered back to this graph structure by the compiler when it translates your code to runnable byte or machine codes (even if you don’t think about it).

But how does it help us solving real problems? We need the Dust platform that can execute such plans directly, and the Montru environment allowing us to learn, use and improve such plans in a global shared knowledge repository. The aim of my research is to build a version of Dust without compromises (I have built different kinds of Dust in the past 15 years because it allowed me solving “impossible” problems), and a version of Montru that depends on Dust only (the previous version, Dust Construct was an “old-fashioned” Java Swing application that I needed to solve my own impossible problem of designing the Dust core).

Research Context

I have found impossible to talk about this concept in the past 15+ years, even though I could show how working systems used it. It is very important to put it into context.

First, I am thankful to faculty members here at the University who kept asking me questions and requesting a scientific definition of the difference that tells my work apart from existing projects in this heavily populated area. Finally, I have this definition. I want to create the minimal sufficient set of Domain Specific Languages (DSLs) that can describe any kind of knowledge and process represented in any information system with precision required for execution, including itself because this DSL set is just another information system.

This complete self-containing nature is the key difference from ontology editors like Protégé, UML designers like Rational Rose or Enterprise Architect, or university research projects like Vanderbilt ISIS. We don’t want to use Protégé to change its user interface, we don’t think of changing UML element definitions in EA etc.: the knowledge they hold exists in their source code implemented in an environment and locked as final. I build Montru to edit Montru (or anything else), Dust to execute Dust (and everything else). I want to allow Montru users to discover all the knowledge required to build them and change anything anywhere as they want to.

I have lately found that my aims are quite close to what Douglas Engelbart worked on (from knowledge cards described in Augmenting Human Intellect, to NLS and CoDIAK). I learned that the Doug@50 group plans to create a demo for the 50th anniversary, so I contacted the organizer and joined. However, the above-mentioned approach was fundamentally different from their ideas, so after a few interesting discussions, I have left.

My supervisor, Dr. Botond Bertok recommended a more practical environment to demonstrate the power of this approach by working with highly interconnected documents: laws. Laws are lists of paragraphs, ordered in hierarchical structures, having a lot of reference connections both within the same law and to others. They also change over time; some laws change other laws. This complex altering network of knowledge items is exactly the world I want to work with. We contacted to experts who don’t just use laws but participate in codification. We have discussed various issues like changes over time, reference and hierarchy formatting, bidirectional traversing over links, possible views and services. It was not a real surprise to find that this is almost identical to the aims of Xanadu, “the real hypertext environment” envisioned by another great internet pioneer, Theodore Nelson.

Idea Projection

When we want to solve any problem, we understand a situation on a certain level and have a vision what we want to change. We formalize our knowledge using existing languages, use tools to develop, discuss and execute a plan. In every environment we can rely on a single and coherent toolset like math, physics, material science, production experience. Everywhere except for IT, where we have hundreds of competing and incompatible solutions to every possible problems, which also change every month.

Furthermore, when building an information system, we must work on improve our understanding of a situation, which is "always new", partial and tend to change along and after building the system. Every decision may be wrong along the way, but our current toolchain does not really help changing them later when we have our platforms and tools chosen, prototypes or partial implementations tested, fixed and running. This seems to be an inherent problem that I struggled with along decades. However, in my current research I have found that the origins of information science targeted exactly this area.

Starting with Vannevar Bush, who in his As We May Think article in 1945 stated that human science improvement may stop and collapse under its own weight. Scientists and knowledge workers need an infrastructure that enables them using the global body of knowledge, including any ongoing research on any field, as the extension of their brains. He called this infrastructure Memex as Memory Extension.

Having the first ENIAC under construction this was a wild statement, but only 20 years later, Ivan Sutherland came up with the world's first truly interactive system that enabled a cooperative thinking with a computer in his PhD thesis, Sketchpad.

In another 20 years we already have the prototypes of our current IT environment running in PARC led by Alan Kay: generic modelling tools that allow creating any information systems in a dynamic improvement process, in a global communication network. It also facilitates learning, allows children explore and understand physical or biological phenomena.

When talking about this infrastructure, we must remember Neil Postman, who asked critical questions and warned about individual and social side effects of being able to reach too much information. Current problems like social media addiction or post-truth communication validate his concerns.

Returning to the original question, IT should not provide thousands of tools for the same problem just because it can, but rather focus on what happens before we have our language and tools to describe any problem. We need an infrastructure that allows us to learn the weakness of the terms that we use to describe a situation and let us seamlessly improve the languages and tools within the same process where we already have our knowledge represented and procedures described.

There is a great analogy here with computation theory. Automata represent our understanding of a task, it is a clean, reliable white box model, but only works when we know this structure. Turing machines focus on processes, so we can deal with environments we don't know – but we can't rely on them, nor hope to understand the task better if they get too complicated. Computer aided knowledge work should use both approaches: use codes (TMs) to set up hypotheses, test them using real life data. Then refactor: extract the knowledge (better structural understanding) to create or improve system architecture, which is a graph representation, a kind of improved automata, to handle the task in a cleaner way.

This iterative, but constantly improving process is what I call idea projection, and as you see, it takes place "before" what we today call knowledge representation and gives a formal environment to the dynamic language and tool adaptation. Let's see how this works in real life development.

Knowledge Broker

As part of the Doug@50 cooperation, I restarted Dust development, and there is a delicate segment that causes a lot of trouble: booting the kernel. Launching an application with an external runtime is OK, you learn how to write “main” and go. But a custom runtime that one day will also be an executable plan and a minimal source code generated from it is another business. It should only be able to initialize a persistent store loader that can create all the components and knowledge the kernel consists of, then load the configuration of the application and launch it.

So, I started writing the persistence component, a simple JSON parser. I soon realized that I want to minimize the dependency on the JSON parser itself (which is responsible for the JSON syntax only), and the heavy weight component is one that tries to understand what comes from the JSON file, technically, statements about entities, where the entities may be type definitions, attributes etc.: the “language” that this source want to talk with the knowledge repository.

This component is totally independent from the JSON parser, so I could move it out from that source. Then I thought about saving information back into the JSON and realized that this component must not be passive. It should know what each side (the JSON store and the knowledge repository) know and ask back on any unknown term used by the sender. When I want to save something into the JSON file, it is not me who should traverse the connection graph, but the broker should ask me back if the JSON does not yet know something that the target object refers to. The same may happen when loading: the knowledge base should ask back to the JSON for any unknown terms. But what happens if the JSON data refers to other sources? I can have brokers connect my center knowledge repository to other sources. I launch my mail client from a JSON file, but it may refer to my display settings stored locally on my mobile phone, and load the contact list, which is stored with my Google account. This all may be arranged in a homogeneous way by only adding these sources to my repository via instances of the same broker component.

Adding a generic user interface to this broker is enough to interact with any knowledge in this environment. Users connect to the knowledge repository by asking questions, following connections, and the brokers will deliver them all information. The same way, to change anything, they need to connect to the relevant sources and learn their languages, otherwise they can’t make any statement related to them. I always thought that the definition of artificial intelligence is when you can’t control but should convince a system – well, this is how that is done.

And to make it more fractal-like, brokers can connect to other knowledge repositories and investigate other knowledge works directly. For example, you are interested in a reptile. There is a research group responsible for maintaining the knowledge about that species, so your request will give you their current stable knowledge as a persistent store. However, you will immediately see that they have several running research projects related to that reptile, so you can choose to see not the official state, but the questions, changes, reports etc. and you can decide which information to use. If you connect to the ongoing ones, the broker will also notify your environment on changes and keep it up-to-date, and the same way, you can recommend ideas to the group. Of course, there is nothing new in this, distributed version control systems work the same way. The difference is that all changes are transparent to the system, it does not see meaningless text files, but knowledge.

The aim of this slide is not to explain the solution, please don’t pretend that you fully understand or expect me being able to plant it into your mind. Anyway, I estimate that this solution is 35% good, and will change a lot when using it, so don’t waste too much time on studying it… I wanted to show you that along this process, I did not have a plan and did not want to solve a problem, but I had a task and wanted to understand its structure.

I used ordinary tools: Java programming language, Eclipse IDE, jsonsimple library (great, highly recommend it) but in a special way to explore possible solutions, write code of minimal size and maximum density, focus on structural and not performance optimization. I refactored continuously from giving the proper names to the elements to shredding and reorganizing complete, already working versions.

The lesson here is that a working solution is not the end of the development, only the proof that you have the minimal understanding necessary to do the job. Then you should start extracting your understanding from the code and make the structure that models the task, separate responsibilities into closed components and implement the required behavior by their interaction. Finally, look at them and see what else they can do. Because if you did a good job, those components will be smarter then you could have planned before, and they let you see the system from a new perspective. You started to build a language that you can use to describe any state and solve any problem in the current domain – instead of executing one and moving to the next until you finish the list of requirements.

This is that dynamic, interactive Idea Projection instead of directed Knowledge Representation, this is that formalized and augmented learning process that we need to explore an information system while making it, instead of either pretending that we can know everything before starting to implement (V-model) or hoping that we can somehow grow a coherent system from solving individual problems without a master view (Agile as executed today).

Law Reverse Engineering

In his book Libraries of the Future (1964), JCR Licklider detailed the requirements and services of a global online information system replacing all scientific libraries by 2000. He initially separated “transferrable knowledge” like scientific literature and photographs from artistic texts and paintings and narrowed the research on the first group because that allows objective evaluation independent from human perception, therefore meaningful automated processing. Such texts should have very complex, cross linked structures revealing dependencies and the thought process of knowledge workers who built them, where arguments have no connection to taste or popularity but pure logical reasoning. Although he described complete reverse engineering of knowledge by parsing the text for knowledge, in a simpler form we can focus on building an environment managing the interlinked network of background knowledge, situation descriptions and plans based on them.

An example of such systems can be built on laws, because they have this complex structure, so we can experiment with how to store and manage them. We should somehow reverse engineer their internal hierarchical structure, then find references pointing to other locations and store them separately to support visualization, navigation and search. Later we should also manage “delta” texts that are there to change other laws and provide a time window showing the current state at any given time or display changes over a specified period. Finally, we can also add services to edit such texts, either directly or by generating the “delta” text nodes. There is also a “meta” information like the hierarchical arrangement and heading numbering that are used when creating references.

Of course, reliable reverse engineering such amount of information is not very likely from a freely formatted text like a Word file. We searched for better sources and found net.jogtar.hu by Walters Kluwer. We tried to contact to the publisher but this far they did not respond, however we decided to continue. We only use publicly available information with minimal load by caching responses and do not plan to provide any service based on their data, only a proof of concept academic research project. The HTML response of their web site both for query and download is very clean and already has structural information, unique identifiers for any text element and heading classes for the hierarchy.

In Java it is very easy to create a web access with local file system cache to create a GUI to search their site, display the response and download a law by clicking on it. With a HTML DOM library (JSoup in this case), it is also easy to investigate the content without limitations. Naturally, for performance reasons a SAX parsing would be better, but without preliminary knowledge on the structure and no real memory limitation, DOM is the better choice for the mentioned structural optimization. Here I can use the information just arrived, like the organization of the headers, class names, containment hierarchy to extract information, like the header hierarchy in the example by proper regular expressions. This way, it is not too complicated to build a knowledge base required to parse any law coming from this site.

The first step is to write the services required to manage this content, like a cloud-based text node and hierarchy storage, the proper data model that allows storing references, lists and trees, or JSoup processors and regular expression evaluators to extract the required information. The quickest way to do that is to play with a selected law text directly in the debugger, and the code will be linked to that law. Then these tools must be separated from their actual content (like the regular expression templates or the numbering schemes), until the point when any law from the same publisher can be handled by simply configuring the components, but not writing any new source code.

Then the last step: make this toolbox available for the system itself: extend the services to contain all plausible configurations and let the loader process deal with the law text automatically, matching the options and find the fitting meta settings for the actual law. Of course, I don't assume complete automatic reverse engineering, but it should take the initial step and allow the user to fix any error in the processing by correcting meta settings or manually process problematic text elements.

The extra feature here is storing the resulted knowledge, both the parsed form of the law and the detected and optionally customized parser configuration (in this case, replacing a complex and easy to spoil source code) with the same tools. In this specific domain, this system "writes its own processor software", not in a platform-dependent source code, but organizing the terms of the law processing DSL we have just created, into the abstract graph representation described at the beginning of the lecture. This is allowed by a "quick and dirty" full-Java Dust implementation but shows the conceivable power of a complete self containing system: the final Dust/Montru environment.

Academic Progress

At our Doctoral School, we should register for an exam in the 4th semester, but only if we have the required lecture and publication credits, the first is now done, but I have no publication. I should have at least 2 conference papers or one accepted article in an international journal by November. I hoped that via the Doug@50 group I can get there somehow, but that was a failure, and switched to the law project too late.

After this warning, I had to reconsider my motivations. I am 45 and have spent quite a lot of time and effort at the University to get my MSC degree and continue with PhD in the past 3 years. I have no ambition to go to conferences or participate in what I see an academic life, I am a problem solver and have a lot to do. Furthermore, my research topic starts with Vannevar Bush, who called for information technology because of the obsolete communication patterns, namely the current scientific publication process. I have spent 20+ years in the industry, I understand the business model of publishers, but I don't have to like it, and I neither have the money nor the motivation to buy my way out of it. I know about Aaron Swartz, I share his ideals without his optimism.

Now it seems that I will suspend my PhD for the next semester waiting for areas where I can get the required credit points, and if that does not work, I will quit.

Documents

When we think about documents, their physical form appears in our minds: nicely arranged rows of characters, layout to separate segments, formatting to indicate important parts. When reading, we get a story, have feelings, derive consequences, agree or disagree with the author's decisions, like or don't like the writing style, pace, wording. All this is essential part of being human and in our culture, but have absolutely no relevance for scientific documents, laws, medical protocols, etc.

"Transferrable knowledge" (by Licklider) is about the following. An author or authors did a research and have a different understanding on their field or organized the existing knowledge in a way that they think would be easier to understand for an audience. They take a subset of that complex network, and "serialize" their statements in an appropriate order. The audience should not think of it as a story, they should rebuild the same structure in their minds; look up any missing elements in the referred or other sources, use or criticize its statements in their own works. In education or at work, the quality of this reconstruction is tested, because the reliable operation depends on proper and common understanding of the background. This is the same for a farmer, a nurse or an engineer (but unfortunately not in mass or social media and politics…)

Therefore, documents should not focus on the layout, but the structure and links among statements, formatted text editors do not support but instead corrupt meaningful knowledge storage and transfer. Strong statement, how can I prove it?

Douglas Engelbart wrote his Augmenting Human Intellect article in 1962, created and demonstrated NLS (oN Line System) in 1968. On the demo, you can see pages of texts and ways to summarize them and navigate around the "document cloud". The topic of the demo was NLS itself, this "cloud" contained all knowledge from the initial requirements down to the sources running on the machines. Later Engelbart refined his approach to CoDIAK: Concurrent Development, Integration and Application of Knowledge. According to this approach, all organized operation has a knowledge repository in the center. If that repository is amorphous and hidden in the participants, emails, documents, information system source codes etc., then the organization can't optimize its operation or adapt to changes. Only a properly organized and flexible infrastructure in the middle allows truly "organic" efficiency and flexibility, conscious improvements. NLS was not a document editor, it was a CoDIAK environment for its own development.

Ted Nelson also focused on the interlinked nature of knowledge segments, that appear as flow texts, but their references, and the ability to wander around them in an online visual environment is the key to properly rebuild the network in the readers' minds. Nelson started this idea with simple drawings, but spent his whole career trying to build this model, Xanadu. Although this appears as a failure, some may know that the first working Hypertext Editor System (HES) was created at Brown University by Andries Van Dam in cooperation with Ted Nelson and was used to document the Apollo Space Program.

Another strong statement against formatted text editors is that the required one is already here: TeX by Donald Knuth. It clearly indicated that really required features of text formatting are limited and having one system develop in good hands without the need of generating profit gradually settle down. Today we have plenty of document formats and different editors changing over time, so all documents created in them are subject of becoming obsolete or unusable without their proprietary editors. Of course, this makes place for more platform-independent output formats. On the other hand, TeX knows everything the scientific community needs to store knowledge in a formatted and well lay out, editable, platform independent and permanent form.

However, that format still lacks the structure, as researched by philosophers like Tim Van Gelder. He focuses on how to extract the argument structure back from the documents, how to look for errors and on the other hand, how can we augment the writer to create structurally clean documents. No surprise that he refers to Douglas Engelbart and his research.

If the importance of a document editor focusing on the structure is so evident, what is missing?

I think the first and most important finding is that the atomic element of a text is not the character, but the referable statement. When creating a document, we don't write it line by line, but create a cloud of initial ideas (statements) and add statement references from other sources. Then in a parallel operation, edit their contents, connections, set up a reading order and heading hierarchy.

References should point to these statements instead of character ranges, consequentially they will survive any change above the referred part, reorganization of the remote document hierarchy or translation of the text, because that has the same text id but different the language code.

In this arrangement, the same statement may appear in different documents, which are custom arrangements of statements. Consequentially, the reference can contain not only the statement internal identifier, but the hierarchy node by which it should appear (this works for displaying law references for example). The system should also consider hierarchical arrangements outside (headings) and inside the text body (bulleted, numbered, multi-level lists) as well.

There are special elements like definitions (a longer text about a term, this may appear as index or in a popup when the term appears in the text), meta (like "Chapter 1", which is in fact not plain text but generated from a language dependent template, this template is meta information in the document), Reference (in various forms) and Delta (a text that describes a change in the referred target text element).

This storage model allows improved user interfaces and services like showing the connections among text elements, browse by following relations, or editors that generates delta text fragments and save them in persistent form (that allows a persistent history management automatically, far beyond "track changes").