Speed up the Conception of Logical Systems with Test-Driven Development

Mathieu Vidal

Institut Jean Nicod (IJN)
Pavillon Jardin, Ecole Normale Suprieure
29 rue d'Ulm, 75005 Paris, France
math.vidal@laposte.net
(+00 33) (0)4 85 02 08 59

Abstract. In this paper, I stress the utility of employing test-driven development for conceiving logical systems. Test-driven development, originally invented in the context of Extreme Programming, is a methodology widely used by software engineers to conceive and develop programs. Its main principle is to design the tests of the attended properties of the system before the development phase. I argue that this methodology is especially convenient in conceiving applied logics. This technique is efficient with tractable logics owning a software implementation. By possessing a clear list of the wished inferences right from the beginning, it is possible to refine step by step the properties of the system until its achievement. I distinguish and detail seven advantages of test-driven developments usage for the conception of a logical theory. In particular, this methodology increases the predictability of its inferential power. A second important benefit is that the time consumption for the conception of the logic is going to go down dramatically. This methodology pushes aside some habits. Logic is no more considered as a formal but as an empirical science. The researches in the field are geared toward some practical and concrete goals. Programs are considered as tools to verify the conformity of the formal system. Finally, I defend the view that in some situations, this way to design a formal system brings strong benefits and that the construction of new logics could be conducted more in line with the development of open source software.

Keywords: Methodology, Test-Driven Development, Applied Logic, Empirical Logic, Logic Engineering

1 Introduction

What are the steps involved in the construction of a logical system? This question has few answers in the available literature. Textbooks and articles generally present logic in its final form. The system is already constituted by the classical trinity of language, proof theory and some semantics which are united by the soundness, completeness and decidability theorems. However, what are the methods employed by the logician to conceive a system? Perhaps logic is an art that

a beginner can only learn with a mentor. In that case, there would be no recipe, process and cookbook to perform research. Only intuition, which is improved year-on-year, would guide the logician during his quest. Another close option would be evident methodology. Continual practice of exercises would lead to the necessary skills required to achieve a system. Nevertheless, even if the number of logicians grows continuously, the reflections concerning the techniques and methods employed in the field are too scarce and are not sufficiently shared between their members.

The central goal of this article is to present a method that can tremendously help the logician during his work. This technique, called *test-driven development*, a popular process used in software development, automates the tests of a program before its development, ensuring better productivity and increasing the final quality of the product. As a particular logic can often be transformed into a program, I argue that it is possible to employ this technique at the middle stage of the inception of the system. The first section of this paper offers a detailed presentation of test-driven development. The second section shows when it is possible to apply test-driven development to the conception of a logical system. The third section presents a concrete application of this methodology. The fourth section details the benefits of this usage. The fifth section examines the links between logic and computer science. Finally, some possible improvements concerning the presentation and development of logics are displayed.

2 Test-Driven Development

The least we can say about IT projects is that they are not always successful. Worse, some reports [1] indicate that the average perception of their failure is greater than 50 percent. From its beginning, IT industry searches to improve this ratio by adopting methodologies that are more efficient. Extreme Programming, which appeared at the end of the 1990s, is such an attempt. Its creator, Kent Beck gathered some of what he considered as the best practices of the field. Test-driven development, or TDD for short, is one of its key concepts [2]. The idea is to reverse the usual order of the cycle that composes software production. The traditional approach is to use a V-model. First, you design. Second, you code. Third, you test. On the contrary, TTD requires that after having defined the design, you must code your tests before the beginning of your implementation. So, as soon as you understand what the goals of your program are, you must write the tests that will assess the success or the failure of your coding.

As part of Extreme Programming, the tests are even a replacement of traditional design. The software must not be conceived and described in complex documentations. On the contrary, short-term cycles, where the new features will be described mainly by test cases, are performed. This approach claims to have the following advantages. It allows a quick adaptation to the changes of business requirements. Furthermore, as the work of the members of the team is divided into short units, its advance is more manageable. The coding tends

toward simplicity. Finally, the software receives only the features that are useful and measurable.

In practice, the writing of the tests is not really the first step. In fact, the programmer starts by developing the interface of his new feature. This piece of code describes the functions and methods that must be achieved by listing their names, inputs and outputs. The interface is often described as the contract between the customer and the service. Then, the developer writes the tests, which try to cover all the different cases of execution, comprising ideal situations and the ones that will trigger errors. The goal is to evaluate the behaviour of the piece of code in a large range of scenarios. All the tests fail, as implementation of the interface is not carried out in this stage. In the last stage, the writing of the execution code is done. The programmer develops the methods or functions one by one, until all the tests are successful. The process can be considered as complete when the features of the product that were specified at the beginning are obtained and successfully tested.

In general, modern programming languages possess specific API specialized in testing capabilities. For instance, Junit [3] is the most well known test library for Java. Modern build tools have also dedicated tasks for testing. Always in the Java ecosystem, Ant [4] allows a user to integrate a step of testing inside the build. Finally, Maven [5], which is more recent than Ant, owns by default a test phase during its whole process of integration. Even if the complete methodology of Extreme Programming is rarely adopted, the focus on early and automatic tests is now widely shared by most IT projects. In general, TDD is emphasized for unit testing. It means that one test concerns only a unique class and more precisely, only the features that this class implements. But TDD can also concern integration phases. In that case, it is not just one class but a whole program or a complete module that is under examination. Some programs are more adapted to this kind of testing. For instance, it is easier to test the result of a file processing than to verify the changes on a complex IHM.

3 Is Logic a Formal or an Empirical Science?

Logic is mainly seen as a formal science[1]. After all, modern logic was born within mathematics during the 19th century. Furthermore, Boole and Frege, the founding fathers of logic, claimed to have discovered the universal laws of reasoning, not from experience but from an *a priori* point of view. Boole[2] considered that a law of logic is manifest in each of its particular instances. Therefore, an induction based on a collection of observations is useless in obtaining these general laws. Frege was opposed to psychology and argued that the entities of logic and mathematics exist independently[3] of our world. Then, the description of logic was made mainly from axioms and rules of transformation, which were posed as

[1] For an examination of the different significations of formal logic, see [6].

[2] See [7], section 4, chapter I.

[3] They belong to a third realm, different from the realm of the things of the outer world and from the realm of our psychological ideas. See [8].

intuitively evident. As soon as the paradoxes of set theory were resolved, whatever were the counter-intuitive consequences of classical logic, it was considered to be the pure logic, by opposition to the corrupted logic employed in the natural language.

This initial development, immune to the constraints of the experience, allowed designing the mathematical tools that were appropriate to this rebirth. Numerous researchers united their efforts and produced a quick growth of this field. But despite its success, especially in mathematics, the new logic soon reached some limits. Most of the reasoning that are present in the natural language and are employed for tasks that are more practical are not correctly modeled by classical logic or are beyond its power of expression. Two main remedies exist. One can try to extend classical logic or one can attempt to revise it. The result of both attempts is the explosion of the number of existing systems.

As soon it was recognized that a unique logic does not exist but only numerous possible formalisations, the respective evaluation of each system becomes relevant. Several criteria to assess the merit of scientific theories exist. But for an applied theory, the correspondence between the predictions of the system and the experimental data is considered as the most important feature. For logic, it means that we need to assess the correspondence between the validation of inferences by the theory and the intuitions we have concerning these inferences in practical cases. A further precaution must be employed here. Depending on the context, the validity of some inferences can vary. In that case, we must fix the field of application of the logic at task precisely. So, one possible inquiry for the logician is the design of a system that is aimed to validate and invalidate a precise list of inferences. Here, logic is seen as an empirical science, where the methodology of TDD can be employed. It is possible before the conception of the system to code the tests that represent the different schemas of reasoning and their expected results.

This method is not applicable to all logical systems. First, some logical enquiries are not directed toward practical issues or for obtaining results that are not easily testable by computational means. These types of researches are not concerned with TDD. Another desideratum is the production of an algorithm from the system, that is, the logic must be decidable. Furthermore, the decision procedure must be tractable. So the problem necessitates a polynomial-time solution, or a non polynomial-time solution but with a very small number of input data. As TDD enforces the computation of numerous inferences, the result of each calculation must be proceed in a short time. Without this property, TDD will offer less help. It can always be used to test inferences but the results can be too long to obtain. Finally, to avoid any confusion, I want to make clear that a logic unsuited for TDD can be extremely interesting. You do not have to trade your initial ideas for tractability. In that case, TDD is just a methodology not adapted to your needs. But if one of your goals is to obtain an implementation on computer, TDD could probably be helpful.

TDD is not a panacea. It is not relevant for the primary steps of conception. It cannot provide the basic elements of the theory, as the general features

furnished by the syntax, semantics and proof theory. During this early phase, the system is not sufficiently mature to receive an implementation. So the initial design will certainly be conceived without the help of TDD. But as soon as the system is sufficiently promising, the implementation can start. From this moment on, TDD will be of tremendous help. In fact, a decidable logic can be seen as a type of software that is especially simple to test. Its inputs are strings of characters, which represent the inferences. Its outputs, the validity or invalidity of these inferences, are Boolean. We can also choose to discriminate between valid, consistent and contradictory sentences. In that case, the outputs will be some integers. Anyway, the return values are extremely easy to analyze by computational means.

4 A concrete case

4.1 The usual way

To let the reader better grasp the difference between the normal way to implement a theorem prover and the use of TDD, let me tell you a concrete application of this methodology. I employed it during the design of a conditional logic, which is detailed in [9]. The first part of my work was conducted as usual. I started from some intuitive ideas concerning the underlying meaning of this type of reasoning, devised a proof theory and provided a computer implementation. In this logic were defined six different conditional connectors and two different contexts of use. To be able to compare my system with concurrent ones, I implemented tests verifying the validity of a great number of inferences. At this point, I obtained twelve (6 connectors times 2 contexts) classes of test, with around 50 inferences evaluated in each class. Finally, I searched an example from the natural language for each inference. Here is the beginning of one of my Junit test classes. The symbols employed are '#' (syntactic consequence), '&' (conjunction), '$\sim$' (negation), 'v' (disjunction), '>' (first intensional conditional) and '-' (material conditional).

```
...
public class InferentialConditionalTransitiveBivalentTest
extends TestCase {

/** Object which tests the validity of the inference. */
private Validator validator;

/**
 * Preparation and definition of the implementation of the validator.
 * Here the validator is in a transitive context
 * (the constructor use the argument 'true').
 * @throws Exception the exception
 * @see junit.framework.TestCase#setUp()
 */
```

```
protected void setUp() throws Exception {
super.setUp();
validator = new Validator(true);
}

/**
 * Basic propositional deduction.
 * @throws Exception the exception
 */
public void testPropositionalDeductions() throws Exception {
assertEquals(true, (validator.validateDeduction("p#p")).isResultat());
assertEquals(true, (validator.validateDeduction("p&~p#q")).isResultat());
}

/**
 * Classical inferences in conditional logic.
 * @throws Exception the exception
 */
public void testClassics() throws Exception {
// Modus Ponens
assertEquals(true, (validator.validateDeduction("(p>q)&p#q")).isResultat());
// Modus Tollens
assertEquals(true, (validator.validateDeduction("(p>q)&~q#~p")).isResultat());
// False Modus Ponens
assertEquals(false, (validator.validateDeduction("(p>q)&q#p")).isResultat());
// Material Conditional to Intensional Conditional
assertEquals(false, (validator.validateDeduction("p-q#p>q")).isResultat());
// Conditional Excluded Middle
assertEquals(false, (validator.validateDeduction("#(p>q)v(p>~q)")).isResultat());
// Strengthening of the Antecedent
assertEquals(false, (validator.validateDeduction("p>r#(p&q)>r")).isResultat());
...
```

4.2 The TDD way

At the end of the conception of this first system, I asked myself if I could improve it by designing a connexive logic. This type of system validates the following inferences (see [10] for a general presentation):

(AT) $\# \sim (\sim p > p)$
(AT') $\# \sim (p > \sim p)$
(AB) $\# \sim [(p > q) \& (\sim p > q)]$
(AB') $\# \sim [(p > q) \& (p > \sim q)]$
(BO) $p > q \# \sim (p > \sim q)$
(BO') $p > \sim q \# \sim (p > q)$

The semantics and the proof theory of the first system were based on a decidable fragment of set theory (see [11] for an entry point). My first intuition was that I had to expand my semantic conditions in the following way. The antecedent and the consequent could not be respectively an empty set and the set universe. This idea was conclusive when the elements of the conditional were atomic. But I had no immediate means to decide how to apply it to complex sentences. For instance, with an antecedent of form 'p and q', must I require for each atom in turn to not be an empty set? Must I require this condition either for the complex formula alone or for the atoms and the compound sentence? By playing with this idea, the number of slightly different variations was growing quickly. But the few manual derivations which I proceeded were unable to show any difference between these options. An important desideratum was also to conserve as much inferences as possible from the first system. I had already searched examples from the natural language that I found convincing. I did not want to change these inferences that I consider extremely intuitive.

My first option was to proceed as before: compute manually inferences until I discover manifest incoherent results. However, without evident anomalies, I had potentially to compute all the inferences. Remember that I had six connectors, two different contexts and around 50 inferences for each combination. It means that I had 6 x 2 x 50 = 600 derivations to proceed. By taking 5 minutes to compute and report each one, the average time of evaluation for one option would have been 50 hours. By evaluating three options, I had potentially to take 150 hours for this work. And I was quite certain to make a mistake during the process.

It is here that TDD enters the stage. From the implementation of my first system, I had already the campaign of tests that delineates the wanted results. I just implemented the new semantic ideas in my software and changed slightly the interfaces to be able to select the connexive implementations when needed. I copied also the test classes, modified the attended results for inferences AT, AT', AB, AB', BO, BO' and applied them to my new connexive connectors. This implementation for the three options took me two hours. At the end, I could quickly compare the inferences obtained in each case. In fact, the processing time for each option took around six minutes on my old computer (2.08 GHz, 512 Mo). Here is an instance of the outputs produced by using Maven. As several inferences are grouped in each test, behind the 170 tests are computed around 1000 inferences.

```
>mvn test

....

Results :

Tests run: 170, Failures: 0, Errors: 4, Skipped: 0

[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5:51.937s
[INFO] Finished at: Sat Jul 16 10:25:24 CEST 2011
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

Unsatisfied, I finally tested a fourth option: the literals of the antecedent and of the consequent could not be respectively an empty set and the set universe. This last possibility was conclusive. For my first conditional connector, I loose only one inference relatively to my first system: the identity. To invalidate the schema 'A >A' was acceptable under my new semantic conditions: the literals of A could be tautologies or contradictions, invalidating the requisite to not be an empty set or the set universe.

To resume, starting from a general intuitive idea, I discriminated between the possible variations by empirical results in a short time. This evaluation was conducted by the use of test campaigns, as advocated by TDD.

5 Advantages of TDD

In this section, I present the different advantages arising from the use of TDD. They are grouped under seven high-level characteristics.

5.1 Gain of time

Even with a general idea of the proof theory to be employed, some minor choices are often required to achieve the system. The first virtue of TDD is to test one by one these modifications and have quick results. More test inferences we have, greater will be the time gained. To compute the proofs one by one with the pen is tedious work. Furthermore, we can commit quite easily errors. For instance, if we have to test twenty possible choices of implementation against two hundred inferences, this work will take days. Contrarily, by launching the automatic test campaign, we will obtain results in just a few seconds or minutes. Certainly, there is an initial cost of time to develop the software. However, this effort is soon rewarded with an acceleration of the researches. Furthermore, the development can often be built on preexisting software. Some classical logical systems are already automated and are available for further use.

5.2 Non-regression

One important benefit of TDD is the constant control of the progress of the system. During an advanced phase of conception, one often encounters unexpected results. These unexpected features are too complex to be anticipated right from the start and some of them are not welcome. In such a situation, minor improvements can be tried. One advantage of TDD is to insure that the new versions

do not introduce regressions. In fact, a change can solve an issue but can also cause some problems in other parts of the theory. TDD allows to control in such a way that the features that were correctly modeled are not lost in new versions of the system. I encountered myself this case as Andreas Herzig suggested me to use a more powerful rule of inference for the modus ponens. I could implement it and judge quickly if its employ was safe or not.

5.3 Separation of Concern

Efficient tests are organized to proceed at different levels. Some of them control the general behaviour of the system but others specialize on smaller parts. A logic is constructed from symbols that own their particular semantics and rules. Unit tests can be designed to assess that each component exhibits the right functionalities. This step is useful to understand precisely the expected role of each piece of the system. For a propositional logic, the rules associated with each operator are the obvious candidates for these atomic functionalities to be tested. Sometimes, it can diminish the complexity of a part. If it is possible to obtain the expected behaviour with an easier implementation, then the theoretical definitions can be simplified. Finally, with tests at higher levels, TDD allows to quickly detect surprising interactions between the basic elements of the system.

5.4 Volume of Data

As already described, a manual control of numerous inferences is painful. On the contrary, with an implementation, the number of tests is not a limitation. The coding of each test is quick. Furthermore, its computation time is very short. It is evident that a system with such an automatic tool will be tested against a far larger size of inferences than a system that is built only on initial intuitions. We can consider that each inference constitutes an experience. For instance, each connector in the conditional logic that I designed is tested against around 50 inferences. This already offers a precise panorama of the inferential power of the system. With TDD, the theory will possess a very large base of experiments and a better resilience to paradoxes. At least, after the implementation of several test campaigns, the researcher will have a better vision of the advantages and defaults of his system. Also the apparition of new questions concerning the behaviour of the logic, which were not initially envisaged, is better handled. In fact, experimentation can be quickly designed and processed. Finally, each new version of the system can be tested against a growing number of inferences. This gives a virtuous circle concerning the development of the theory, which becomes a more precise model as long as the number of tests increases.

5.5 Performance Tests

As soon as the quest of the researcher is an applied logic, the performance of the system is a feature that needs to be evaluated. One option could be to wait

until the completion of the theoretical aspects before improving the processing time. This method is not efficient. Some theoretical decisions condemn the implementation to never be able to accomplish its tasks in a reasonable amount of time. In that case, these choices must be detected as soon as possible. The best way to do it is to have an implementation early and to practice performance tests right from the beginning. Furthermore, theses tests can also serve to discriminate between concurrent theoretical improvements. Faced with two possibilities that offer the same functionalities, we will retain the one with the fastest implementation.

5.6 Memorization of the theoretical choices

Another advantage of TDD is to keep track of the evolution of the system. The code implementing each new version can be stored and reused if needed. Furthermore, the tests associated with a particular implementation can be conserved. They offer a precise picture of the characteristics of each version. In fact, they describe the inferences that are valid or invalid. These lists of inferences can be kept within a particular campaign of tests. They constitute a part of the documentation associated with an intermediary step of the achievement of a theory. By coming back later to the problem at hand, the researcher will find quickly the reasons on why he took such path and let down other options. Even for a continuous research that was stretched for some weeks or months, it is important to keep track of the different development phases of the system. The tests are certainly the most concrete data assessing these different choices. To come back again to the concrete case that I presented, by inspecting the test classes of the abandoned options, I am able to find again what are their validated inferences.

5.7 Assessing progress and achievement

Finally, an important question for a researcher is to determine when to stop his inquiry. Some would say that one must stop as soon as one can publish a paper! But even this goal is not sufficiently precise. When can I be sure that I attain a sufficient number of my initial objectives? Defining at the start a list of tests that describe the wished inferences is a very good way to measure this achievement. During the whole research, the progress can be evaluated by the successful completion of more and more tests. The system can also be easily compared with concurrent ones, which are known to present such successes and defaults. As soon as the theory performs better than the older ones, it can be considered as sufficiently mature to be published. Finally, the researcher can wait until the system resolves all the issues that were at stake at the beginning. In that way, the system designed is surely sufficiently interesting to be communicated.

6 Logic and Computer Science

6.1 Computer system as a tool for Logic verification

Usually, logic is basic to the formal verification of programs in computer science. Proof-based verification and model checking are the two main methods employed to assess that the description of a computer system meets its specification (see [12]). Both techniques are grounded in mathematical logic. Proof-based verification computes a proof for each property at hand. Model checking uses formulas in temporal logic to represent state transitions. In the approach advocated here, the perspective is reversed. With TDD, the theorem prover and the tests are the basic tools that help to design logic. The computer program is considered as a means to verify that the logic displays all the attended properties. From the slogan "logic as a tool for program", we arrive to the catch-phrase "program as a tool for logic". However, TDD unifies even both views. As the logic is tractable, it can almost be identified with its implementation program. So the studied logic becomes the computer program to study. Finally, one important aspect of TDD to not forget is that the methodology is dynamic. The normal process is to encounter failed tests at the beginning. By successive refinements, the logic is improved until passing successfully the whole test campaign. Then the functional system behavior is achieved and the logic fulfils all its intended properties.

6.2 Logic Engineering

In [13], p. xiii, Areces describes logic engineering by the following sentences:

> How do we decide what the best formalism is for a given reasoning or modeling task? Or even more, what are the important rules to take into account when designing yet another formal language? How do we compare, how do we measure, how do we test? These are the questions that the young field of Logic Engineering is supposed to investigate and, if possible, answer.

I agree with this definition but not always with the way this program of research is conducted. This approach can be divided in two parts but only the first one, the theoretic side, is usually put forward. This first aspect studies the general properties of formal systems and among them expressiveness, completeness, interpolation, decidability and tractability. It boils down to the study of the relations between some high-level logical characteristics. The second part of the program, the practical side, which is present in Areces' definition, is usually completely forgotten. It tries to insure that the logic is really fitted to the concrete problem at hand. This aspect necessitates the use of an engineering methodology and not only mathematical techniques. The goal is simply to tailor step by step the logic until it models correctly the reasoning task at hand. I argue that TDD is the right approach for the practical side of logic engineering, as it can check continuously the accord between the formalism and the concrete reasoning process to model.

6.3 Logic guided by tests

A usual way to intend the design of a formal system is to find an intuitive idea that seems interesting to develop. However, why does a logician select some ideas contrary to others as more fruitful? Often, the chosen option seems sufficiently promising to resolve a difficulty of past systems. The new trend could improve the model of a concrete inference, comparatively to previous logics. For instance, standard deontic logic offers a formal system dealing with the notions of obligation and permission. But this system owns well-known limitations, labeled as *the paradoxes of deontic logic*. One goal of the logician could be to resolve these issues. So, the intuition that guides the researches is often aimed at reducing the flaws in previous systems. To discriminate between the inferences obtained with TDD is a rational way to compare concurrent theories. Very often, a new logic is evaluated by its innovations and by its mathematical sophistication. Both criteria are relatively objective and easy to judge. But a more long term criteria is the applicability of the system. Then its deductive power is central for the evaluation. TDD allows putting this aspect as one of the primal objective of the research right from the beginning. This methodology is especially interesting as soon as one goal is to obtain an applied logic.

7 Present and Future

7.1 Logical Systems as Open Source Projects

I have argued in this article that the quick implementation of a logical theory, guided by the tests, could offer great help to the conception of a particular logic. This method brings together the design of a conceptual structure and the development of software. Significantly, if one of the goals of the logical theory is a concrete application, for instance in artificial intelligence, this methodology is completely justified. One additional question is whether some other practices, issued from computer science, can be used to facilitate logical researches. I think so. One way to present research is to publish it. But another way is to give access to the implementation of the logic on the Web, which can be distributed as open source software. In that case, nightly builds can offer some tests of the system. Then a person interested can quickly assess if the logic contains the wished characteristics for his work. This type of development facilitates also a mutual collaboration from remote people. Finally, based on these elementary libraries and by considering a logic as an API, researchers can design more complex logical theories with TDD.

7.2 Methodology in Logic

To conclude I ask for a better sharing of the successful methodologies employed in logical researches. Year after year, logic becomes more diverse and is used by a growing number of people. Some innovative ways to conceive systems appear quite regularly. The publication of these aspects of logical inquiry could benefit the entire community.

References

[1] IT Cortex: Statistics over IT projects failure rate. http://www.it-cortex.com/Stat_Failure_Rate.htm

[2] Beck, Kent: Extreme Programming Explained: Embracing Change. Reading, MA, Addison-Wesley, 1999.

[3] Beck, Kent; Gamma, Erich: The JUnit Cookbook. http://junit.sourceforge.net/doc/cookbook/cookbook.htm

[4] The Apache Ant Project. http://ant.apache.org/

[5] The Apache Maven Project. http://maven.apache.org/

[6] Béziau, Jean-Yves: What is "formal logic"? http://www.jyb-logic.org/jyb-form-final-sp.pdf

[7] Boole, George: An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities. Rough Draft Printing, 2010. First published: 1854.

[8] Frege, Gottlob: The Thought: A Logical Inquiry. Mind, New Series, Vol. 65, No. 259. (Jul., 1956), pp. 289-311. First published in the *Beiträge zur Philosophie des Deutschen Idealismus*, 1918-1919

[9] Vidal, Mathieu: Conditionnels et Connexions. PhD. Thesis, IJN, EHESS, Paris, 2012.

[10] Wansing, Heinrich: Connexive Logic. 2010, The Stanford Encyclopedia of Philosophy, Edward N. Zalta (editor). http://plato.stanford.edu/entries/logic-connexive/

[11] Hilberticus: an online abacus for sets. http://http://www.hilberticus.org/

[12] Huth, Michael and Ryan Mark: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, 2nd Edition, 2004.

[13] Areces, Carlos: Logic Engineering. The Case of Description and Hybrid Logics. PhD. Thesis, ILLC, University of Amsterdam, 2000.