About "trivial" software patents: the IsNot case

Jan A. Bergstra[♣] Paul Klint[♠]

[♣] *Informatics Institute, University of Amsterdam*
and
Faculty of Philosophy, University of Utrecht
`http://www.science.uva.nl/~janb`

[♠] *Centrum voor Wiskunde en Informatica (CWI), Software Engineering Department*
and
Informatics Institute, University of Amsterdam
`http://www.cwi.nl/~paulk`

June 13, 2006

Abstract

So-called "trivial" software patents undermine the patenting system and are detrimental for innovation. In this paper we use a case-based approach to get a better understanding of this phenomenon. First, we establish a baseline for studying the relation between software development and intellectual property rights by formulating a life cycle for the patenting system as well as three variations of the software life cycle: the defensive patent-aware software life cycle that prevents patent infringements, the more offensive patent-based software life cycle that aims both at preventing infringements and at creating new patents, and the IPR-based software life cycle that considers all forms of protection of intellectual property rights including copyright and secrecy.

Next, we study an application for a software patent concerning the inequality operator and a granted European patent on memory management. We also briefly mention other examples of trivial patents. These examples serve to clarify the issues that arise when integrating patents in the software life cycle.

In an extensive discussion, we cover the difference between expression and idea, the role of patent claims, software patents versus computer implemented inventions, the role of prior art, implications of software patents for open source software, for education, and for government-funded research. We conclude the discussion with the formulation of an "integrity axiom" for software patent authors and owners and sketch an agenda for software patent research.

We conclude that patents are too important to be left to lawyers and economists and that a complete reinterpretation of the patenting system from a software engineering perspective is necessary to understand all ramifications of software patents. We end with explicit conclusions and policy recommendations.

Disclaimer. This work was carried out as part of the project *Study of the effects of allowing patent claims for computer-implemented inventions*, a joint study initiated by the European Commission and carried out by MERIT (University of Maastricht, Netherlands), Centre of Intellectual Property Law CIER (University of Utrecht, Netherlands), Centrum voor Wiskunde en Informatica (Amsterdam, Netherlands), Telecommunication Engineering School at the Universidad Politécnica de Madrid (UPM), Spain and Centre for Research on Innovation and Internationalization (CESPRI) at Bocconi University, Milan, Italy.

The opinions expressed in this publication are those of the authors and do not reflect in any way opinions of the European Commission or any of the partners in the above mentioned consortium.

1 Background

For many years, there have been concerns in the Unites States (US) about the possibilities to patent "trivial" software techniques and business methods. The patenting laws in the European Union (EU) have always been more restrictive than their US counterparts, but in the discussion about the recently rejected EU directive about patenting computer implemented inventions (CII), or software patents for short, the level of triviality of a software patent has become a focal point in the debate: does a patent lay claims on techniques that are generally considered to be common knowledge or does the patent claim a real invention?

One should be carefull with the term "trivial patent" itself. Informally, it means a patent that describes a small but insignificant advance over the state of the art, but in a strict, legal, sense it means "novel, but obvious" (US) or "novel, but lacking an inventive step" (Europe). As we will see, many software patents that are usually called trivial are not even that: they are non-novel and are anticipated by prior art.

As part of a 3 year European Commission (EC) study on the effects of software patents on innovation we are involved in a multi-disciplinary effort to understand the effects of software patents. These effects are studied from legal, economical, and computer science perspectives. The goal of the current paper is to study trivial software patents from a computer science perspective and to make a contribution to the discussion among experts from the three disciplines just mentioned. For economic effects we refer to [21, 11] and for legal aspects to [32, 5].

Software patenting is a relatively new topic for both authors, as it probably is for most software engineers and computer scientists. For completeness, we mention that both signed a petition to the European Parliament [33]. The second author has acted as speaker on a conference about the topic [13] (later adopted as point of view of the Royal Dutch Academy of Sciences) and has written a column about it [24]. Our professional interest in the topic stems from a long involvement in software engineering research ranging from study of the software life cycle [10], concepts of programming languages [9], theory, design and use of software components [6, 7, 8], generic language technology [12] and program analysis [23]. Both authors have cooperated in setting up the MSc curriculum in Software Engineering at the University of Amsterdam, now organized in cooperation with Mark van de Brand of the Hogeschool van Amsterdam and taught in cooperation with Hans van Vliet from the Vrije Universiteit in Amsterdam. Software patenting is therefore a major concern for us.

The plan of the paper is as follows. First, we start exploring how what seems to be a huge distance between the world of patents and the world of software engineering can be bridged. First we design in Section 2 a life cycle for the patenting process and next we make a connection between patents and software engineering by designing a patent-based software engineering life cycle (Section 3).

Given this conceptual framework, we study recent examples of software patents in order to get a better perspective on the implications for these software life cycles. In Section 4 we describe a recent patent application that might be a candidate for the predicate "trivial software patent". In Section 5 we present various views on this application. In Section 6 we briefly analyze a European patent on memory allocation and conclude that its novelty is strongly debatable. Next, we mention in Section 7 other trivial patents, both from the US and from Europe. A discussion (Section 8) and conclusions (Section 9) complete the paper.

2 The patent life cycle

It is important to describe the phases of the patenting process in such a manner that they become recognizable for the software engineer. We conjecture that the *Patent Life Cycle* shown in Figure 1 is a fair representation of this process. It consists of the following phases:

- An applicant *applies* for a patent.
- The applicant can decide to *withdraw* the application.
- The Patent Office can either *grant* or *reject* the application.
- The applicant can *appeal* against this decision and a reject decision may be changed into a grant decision.

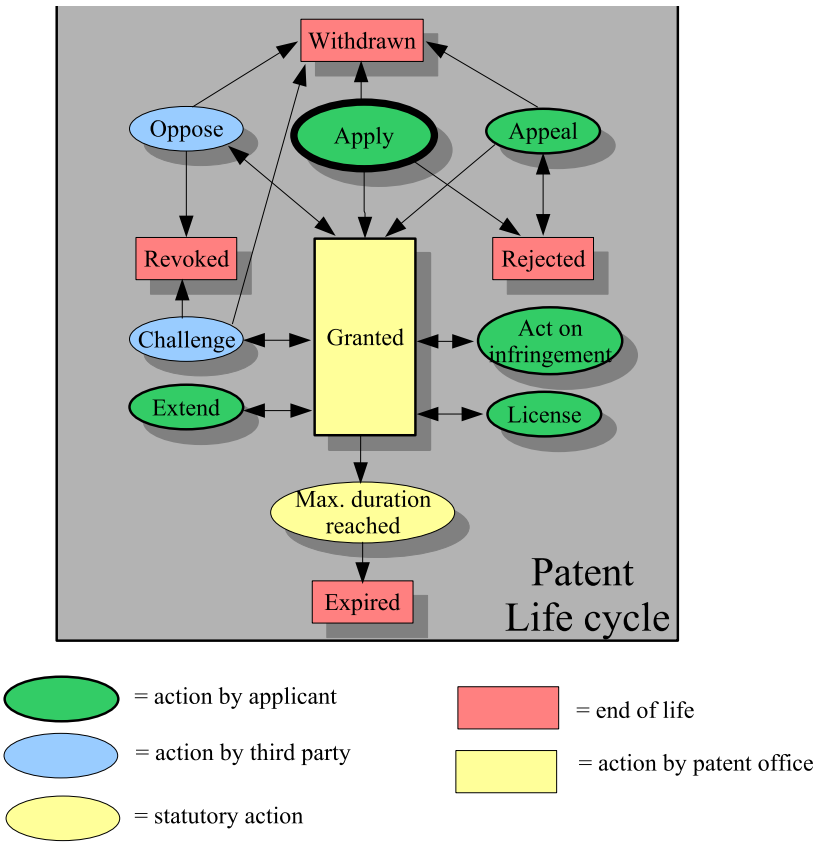

Figure 1: The patent life cycle: from filing to expiration

- The applicant of a granted patent applicant becomes the holder of the patent.
- A granted patent may be *challenged* by another party. This may lead to revocation of the patent.
- The patent holder may act on *infringement* of its patent.
- The patent holder may *license* its patent to another party.
- The patent holder may *extend* its patent periodically.
- The patent *expires* after a maximal duration.

It is open for debate whether this abstraction of the patenting process can be used in the EU as well as in the US and Japan. However, since software developers have to be aware of potential patent infringements, independent of the source of the patent, such an abstraction of the patenting process is essential. This is relevant for developers of both commercial software and open source software.

The IsNot patent to be discussed later on in Section 4 is in the application phase, for all other patents mentioned in this paper we have explicitly indicated their status.

3 Baseline: an IPR-based software engineering life cycle

The next step is to make a connection between the patenting process—or rather Intellectual Property Rights (IPR) in general—and software engineering practices.

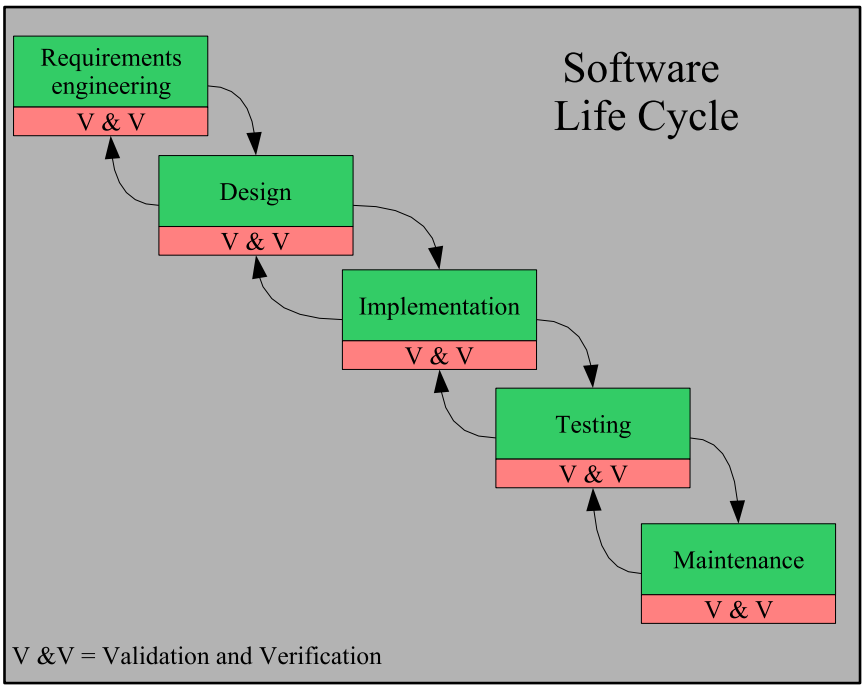

Figure 2: The software life cycle

3.1 The software life cycle

In software engineering, the software life cycle is a frequently used manner of organizing the software development process. Figure 2 shows a strongly simplified version of the life cycle taken from a standard textbook [37]. It consists of the following phases:

- Requirements engineering: collect the requirements and expectations from the future owners and users of the system.
- Design: translate the requirements in a specification that describes the global architecture and the functionality of the system.
- Implementation: build the system.This amounts to transforming the design into software source code.
- Testing: test that the implemented system conforms to the specification.
- Maintenance: install, maintain and gradually improve the system.

It should be emphasized that the software life cycle covers design and construction of a software product as well as its use. Each phase contains a Validation and Verification (V&V) sub-phase in which the quality of the deliverables of that phases are controlled. Also note the backward arrows that make this into a real "cycle": it is possible to discover in later phases that decisions made in a previous phase have to be revised.

We will now proceed in three steps. First, a defensive Patent-aware Software Life Cycle is sketched that ensures that the software development organization does not infringe patents of third parties. Next, a more offensive Patent-based Software Life Cycle is described that also considers the options to file patent applications for knowledge that has been generated in each phase of the life cycle. Finally, the IPR-based Software Life Cycle extends the previous one to all IPR options: secrecy, copyrights and patents.

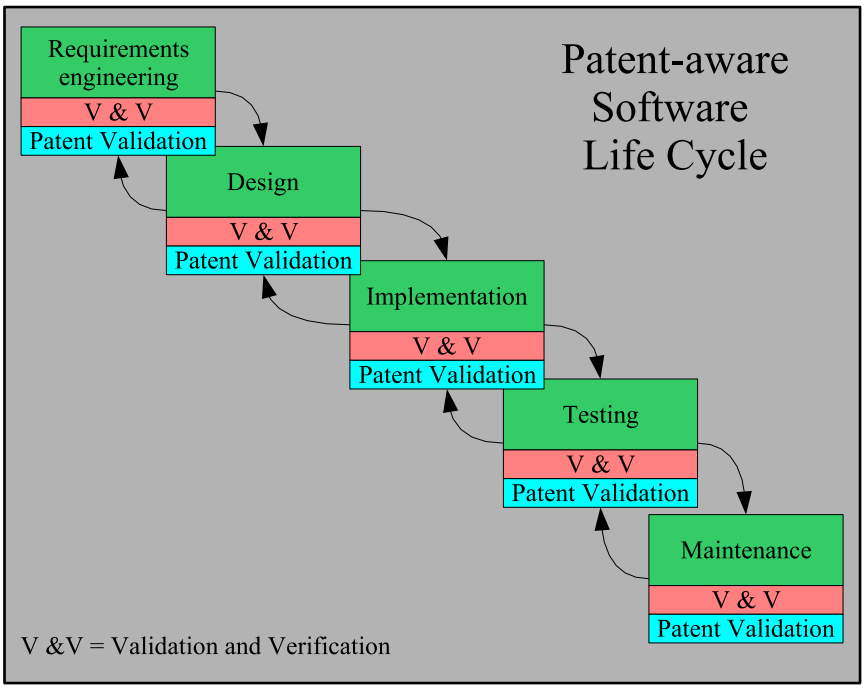

Figure 3: The defensive patent-aware software life cycle

3.2 The patent-aware software life cyle

In Figure 3, we sketch a Patent-aware Software Life Cycle in which an extra sub-phase is added that performs patent validation. This generates immediately many unsolved questions. For each phase one may wonder:

- Is it possible to infringe patents in this phase?
- If so, how can one find such infringements?
- How can such infringements be resolved?

The Patent-aware Software Life Cycle is a defensive step that any commercial or open source software development process should adopt. Clearly the costs for software development will increase significantly.

3.3 The patent-based software life cycle

It is, however, possible to go one step further. In Figure 4 we sketch a Patent-based Software Life Cycle in which yet another sub-phase has been added that performs patent applications whenever possible. We conjecture that this strategy is only available to the software development organizations with the deepest pockets. For each phase now further questions apply, such as

- Does this phase generate patentable knowledge?
- Should we file a patent application for this knowledge?
- Are there other means to avoid that this knowledge generates an advantage for our competitors?

In many large software development organizations there exist “Chinese walls” between software developers and patent attorneys. This is not only the case for large commercial organizations but also for large open source projects like the Apache Foundation. The rationale being that the less software developers

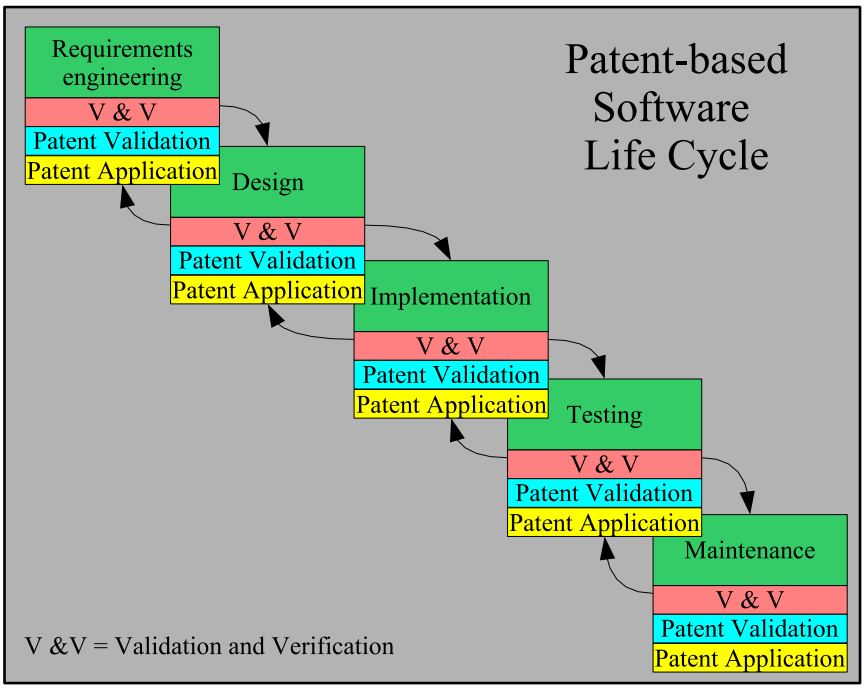

Figure 4: The offensive patent-based software life cycle

know about patents the stronger the position of the organization is in legal disputes. Implementation of the Patent-based Software Life Cycle may require similar measures. Of course, such measures completely defeat one of the primary goals of the patent system, i.e., knowledge dissemination.

3.4 The IPR-based software life cycle

The final step is the IPR-based Software Life Cycle sketched in Figure 5 that takes *all* aspects of IPR into account during software development. For each phase the questions now become:

- Does this phase violate copyrights of others? If so, remove those violations.
- Does this phase infringe patents? If so, negotiate a license with the patent holder or take technical measures to avoid the infringements.
- Does this phase generate valuable knowledge? If so, consider the following three options:
  - Keep the knowledge secret and take appropriate legal or technical measures to achieve this.
  - Establish copyrights on this knowledge.
  - File patent applications for this knowledge.

These questions form a comprehensive description of the IPR policy one would expect of the biggest, multi-national, software development organizations.

3.5 Discussion

These extended software life cycles already raise many fundamental questions that are not easy to answer:

- Is it possible to use these extended software life cycles in such a way that they comply with the major patenting systems world wide?

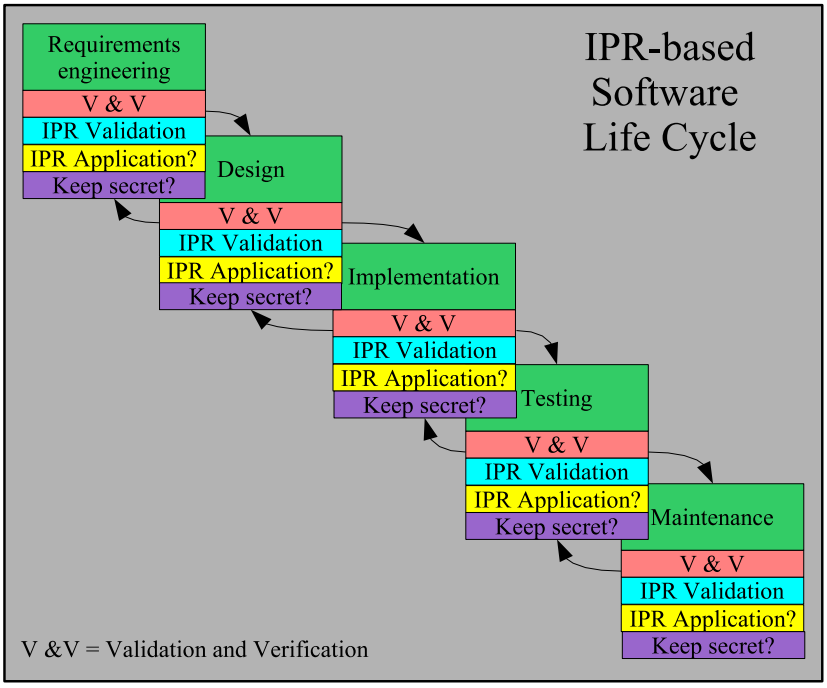

Figure 5: Considering all options: the IPR-based software life cycle

- How can the software engineering knowledge that is hidden in the patent data bases made accessible for software engineers?
- Is it possible to a give an *operational* definition of a patent infringement that can be used by software engineers?
- For each of the phases of the software life cycle (requirements engineering, design, implementation, testing and maintenance) the following questions should be answered:
  - How is knowledge in this phase represented?
  - Where can prior art for this phase be found?
  - How can patent infringements in this phase be identified?
  - How can patent infringements in this phase be resolved?

  We expect that the answers to these questions will widely differ for each phase.
- What are the technical implications for software development when using these extended software life cycles?
- What are the economic implications of the extended software life cycles?

We will come back to these questions in the remainder of this paper and in Section 8.10 we will propose a research agenda.

We will now relate the high-level discussion in the previous sections to the daily practice of software patents by studying several examples.

4 The IsNot patent application

On May 14, 2003 the three Microsoft employees Paul A. Vick jr. (technical lead for Visual Basic), Cosica Corneliu Barsan (member of the Visual Basic compiler team), and Amanda K. Silver (program manager

on the Visual Basic compiler team) filed United States Patent Application #437822 with the title "IS NOT OPERATOR". The abstract of the IsNot patent application (as we will call it) reads as follows:

> *A system, method and computer-readable medium support the use of a single operator that allows a comparison of two variables to determine if the two variables point to the same location in memory.*

The 8 page application consists of 24 claims followed by a description of the background of the invention, and detailed descriptions of illustrative embodiments. The first 5 claims of the application read as follows:

> *What is claimed:*
>
> *1. A system for determining if two operands point to different locations in memory, the system comprising: a compiler for receiving source code and generating executable code from the source code, the source code comprising an expression comprising an operator associated with a first operand and a second operand, the expression evaluating to true when the first operand and the second operand point to different memory locations.*
>
> *2. The system of claim 1, wherein the compiler is a BASIC-derived programming language compiler.*
>
> *3. The system of claim 1, wherein the operator is IsNot.*
>
> *4. The system of claim 1, wherein the compiler comprises a scanner, a parser, an analyzer and an executable-generator.*
>
> *5. The system of claim 4, wherein the source code comprises at least one statement, and the statement comprises a keyword representing the operator, the keyword recognized by the scanner.*

The remaining 19 claims go into more details such as the parser determining that the operator is preceeded and followed by an operand, the fact that error messages are generated when the IsNot keyword or one of its operands are missing, the fact that executable code is generated, and so on and so forth.

The patent application describes that the invention can be used in exemplary computing environments ranging from PC, handhelds, servers, automatic teller machines, and more. The application also sketches in detail the hardware architecture of a typical PC using the invention. The application also explicitly states (in paragraph [0050]) the following:

> *It will be recognized that although in the examples, the operator is designated as "IsNoT", the invention is not so limited. Any suitable case sensitive or case insensitive tag for the operator is contemplated by the invention, such as, but not limited to "Is_Not", "isnot", "Isnot", "Is_Not", "is_not" and so on.*[1]

5 Analysis of the IsNot patent application

5.1 IsNot is a trivial software patent

A lawyer or other non-specialist may be impressed by the clever invention described in the IsNot application, but each first-year computer science student will recognize what it is about: this is the inequality operator between pointer values as is known from many different programming languages ranging from the Branch Not Equal instruction `BNE` in PDP11 assembly language [14] to the not equal operator `.NE.` in Fortran [3] or the not equal operator `!=` in C [22], Java [19] or C# [16].

For a computer scientist, the *idea* of having a single operator for comparing two pointer values is common knowledge and the publications cited above constitute prior art.

[1] It seems that the two occurrences of "Is_Not" are a typo in the application.

For a computer scientist, granting this patent application will have devastating effects since it will cover a large majority of the software worldwide and will completely block any further software development or at least dramatically increase developments costs due to licensing.

In a strict sense, the claims in this patent application are non-novel and they are anticipated by prior art. Colloquially, this would be called a trivial patent.

5.2 IsNot is not a trivial software patent

We find it hard to believe that the highly skilled software developers at Microsoft (or their well-known colleagues at Microsoft Research) are unaware of the prior art mentioned above. It is also striking that prior art occurs in one of Microsoft's own products (the language C#). One is tempted to speculate about the intentions of the applicants and their sponsors with this particular patent application. Several possibilities come to mind:

- They think that the subject matter is new and this should be the default assumption. This raises the question whether there exists (or should exist) a form of "patent etiquette" that assumes that applicants truly consider their invention as new. According to Park [29], there is no explicit duty of disclosing prior art in the European Patent Office, whereas the Patent Offices in the US and Japan require the applicant to disclose the closest prior art that he acknowledges when the patent application is filed. In the US this is done in a separately filed "Information Disclosure Statement" (IDS). See Section 8.9 for a further discussion of this topic.

- They find that the matter of trivial patents and determining prior art need clarification and that filing a patent application is the fastest road to achieve this goal, independent of the likelihood of acceptance. This defines the future options for patenting relatively simple inventions. When rejected, the application builds up prior art and may be used to provide indemnity to clients against intellectual property claims.

What if our first analysis from a computer science perspective is too naive? Is it still possible to discover some form of innovation or hidden meaning in this application that merits its acceptance? We see the following possibilities for this:

- The patent application is about the *specific naming* of the comparison operator. This is suggested by the explicit phrase in the patent application we cited above: "*Any suitable case sensitive or case insensitive tag for the operator is contemplated by the invention, such as, but not limited to ...*". This would mean that the application is not about the idea of an inequality operator but about the specific form of that operator. In this way, the application would establish a form of copyright on the operator name "IsNot".

- The specific context of BASIC is the substance of the application. This also makes finding prior art hard.

- By patenting this specific operator in BASIC, alternative implementations of the language can be discouraged, or at least interoperability is hindered.

- Although claim 1 of the patent application is broader than any of the preceeding three possibilities, the patent applicant may have written claim 1 for the strategic reason of giving the patent examiner an easy claim to reject, thereby leaving the remaining (narrower) claims and providing the examiner with a feeling of having done his job.

- The patent application is not concerned with the IsNot operator or the inequality operator at all. They just serve as a smoke screen to hide an idea in one of the 23 other claims. Is the patent about giving an error message when an operand of the IsNot operator is missing? Is this patent about BASIC-compilers using a certain compiler organization? As far as we can judge these claims describe common practices in compiler construction and language implementation and cannot be considered to be inventions. This does, however, not mean that it is easy to find prior art since most of the claims

are very specific and may not occur in the literature. We invite the reader to investigate these claims and to challenge our analysis.

- The application has yet an other meaning, for instance, challenging the patenting system. In this case, we really congratulate the applicants for their brilliant contribution. Some implications are further discussed in the remainder of the paper.

5.3 Our opinion

Our opinion about the IsNot patent application can be summarized as follows:

- The IsNot patent application would, when granted, lead to a trivial patent and its inventive step does not differentiate itself from the manifest prior art given above. It is hard to understand why this application would be granted. When granted, this patent could indeed be very harmful for further development.
- In a similar fashion as each scientific publication needs a rationale, we miss a rationale for this patent application.
- It is undesirable that others would have the obligation to find prior art. Given the fact that US patent applications are required to disclose prior art, it is at least curious that this application gives none.
- It is unclear what an infringement of this patent (when granted) would mean. Is the design of a programming language that contains an inequality operator an infringement? Is every program that uses an inequality operator an infringement? Is the mere notion of an inequality test in any form an infringement?
- We don't see a convincing argument why a major company would need this patent, apart from tactical considerations where this patent may clearly play a role.
- Is this a typical patent application? It could be argued that this patent application is one of a kind, and that our analysis of it is thus irrelevant. Although we agree that this is one specific example of a trivial patent application, it is an application from a large firm with a large patent practice, and certainly sufficient resources to determine whether an "invention" is trivial, and to identify prior art, prior to submitting a patent application. So we believe that if IsNot may not be a typical patent application, it is certainly *potentially* typical.

6 Analysis of a European patent on memory allocation

We claim that a patent application is part of the patent life cycle (see Section 2) and is thus part of the open literature and should be publicly discussed and scrutinized for novelty and compliance with prior art. One may counter that the IsNot application may very well be rejected. From a European perspective, one may also counter that such an application would never be accepted by the European Patent Office (EPO). Therefore, we will also briefly analyze a patent granted by the EPO that we consider to be debatable.

On June 1, 1998, European Patent #817044 on "Memory allocation in a multithreaded environment" was granted to Sun Microsystems Inc. (US) with Nakhimovsky Gregory listed as inventor. The abstract reads:

> *A method of allocating memory in a multithreaded (parallel) computing environment in which threads running in parallel within a process are associated with one of a number of memory pools of a system of memory. The method includes the steps of establishing memory pools in the system memory, mapping each thread to one of the memory pools; and for each thread, dynamically allocating user memory blocks from the associated memory pool. The method allows any existing memory management malloc (memory allocation) package to be converted to a multithreaded version so that multithreaded processes are run with greater efficiency.*

The 8 page application consists of 21 claims followed by a description of the invention and preferred embodiments. The first 6 claims read as follows:

> *Claims of EP0817044*
>
> *1. A method of allocating memory in a multithreaded computing environment in which a plurality of threads run in parallel within a process, each thread having access to a system memory, the method comprising: establishing a plurality of memory pools in the system memory; mapping each thread to one said plurality of memory pools; and for each thread, dynamically allocating user memory blocks from the associated memory pool.*
>
> *2. The method of claim 1 wherein the step of dynamically allocating memory blocks includes designating the number of bytes in the block desired to allocate.*
>
> *3. The method of claim 1 further comprising the step of preventing simultaneous access to a memory pool by different threads.*
>
> *4. The method of claim 1 further comprising the step of establishing a memory pool for each thread comprises allocating a memory buffer of a preselected size.*
>
> *5. The method of claim 4 further comprising the step of dynamically increasing the size of the memory pool by allocating additional memory from the system memory in increments equal to the preselected size of the buffer memory.*
>
> *6. The method of claim 4 wherein the preselected size of the buffer is 64 Kbytes.*

The remaining 15 claims go into more details about the specific data structure to represent the memory pool and the memory blocks, and about the deallocation of blocks as well as their merger after deallocation.

Although the subject matter of this patent is not as astonishingly simple as that of the IsNot example, any computer scientist will see what this is about: memory allocation as it occurs in operating system kernels and concurrent applications. A simple way to implement this is to have a single pool of memory blocks that can be claimed by one of the parallel processes (threads). However, to avoid corruption of the administration of the memory pool, access to the memory pool has to be strictly sequential. This is achieved by locking and unlocking the memory pool during each request. Since this locking introduces a time penalty, the idea formulated in this patent is to use a separate memory pool per process (thread).

In our opinion the idea to avoid the use of shared variables is trivial. We conjecture that this idea is not new and we think that there is a proof obligation on the part of the applicants to show how this patent improves upon earlier work (see Section 8.9).

This patent can have a major impact on the implementation of most, if not all, operating system kernels and its mere existence poses a threat to further development.

7 Other trivial patents

There are many examples of trivial software patents worldwide.[2] Examples are:

- US Patent 4648067: Footnote management for display and printing (IBM, 1987). This patent describes the handling of footnotes in a text processing system. This is a standard technique that has been used in every text processor since 1970.
- US Patent 5530794: Method and system for handling text that includes paragraph delimiters of differing formats (Microsoft,1996). This patent describes the conversion of text documents from Unix text files to MS Word format by inserting a carriage return character. Since the characters carriage return (CR) and line feed (LF) were invented, different operating systems have used them in different ways to end each line. This is a trivial technique that has been in use ever since.

[2] In this section we give examples of patents which we *suspect* to be trivial in nature and merit further study. A meticulous search for prior art is needed for each example. The fact that a patent is listed here does *not* imply a final judgment on our part of the patent's triviality or validity. The purpose of this section is merely to raise the awareness of potential trivial patents.

- US Patent 5175857: System for sorting records having sorted strings each having a plurality of linked elements each element storing next record address (Toshiba, 1992). This patent describes sorting using linked lists. This is a standard technique found in every textbook.
- US Patent 6877000: Tool for converting SQL queries into portable ODBC (IBM, 2005): This patent describes how SQL queries can be translated into queries for the portable database interface ODBC. This obvious technique must be used by every database system that connects to ODBC.

Many other examples of trivial software patents are known.[3] From the fact that the above examples are all US Patents one might draw the conclusion that the US Patenting Office is more likely to issue trivial software patents. We think that this is not correct. The explanation is rather that the problem of trivial software patents has been in existence in the US for over 20 years and that the US patent databases have received more public scrutiny than, for instance, the European patent database. As an initial proof of this statement we have collected, in a very limited amount of time, the following European Patents that we consider to be trivial. Some trivial, but expired, patents are:

- European patent 10186: Apparatus for handling tagged pointers (IBM, 1980). This patent describes the addition of a tag bit to pointers in order to discriminate them from ordinary data. This is an old technique that has been used in various systems.
- European Patent 97818: Spelling verification method and typewriter embodying said method (IBM, 1984). This patent describes spell checking.
- European Patent 98959: Method for producing right margin justified text data in a text processing system (IBM, 1984).

More recent examples are:[4]

- European Patent 698844: Tunnel icon (IBM, 1996). Describes a tunnel-like icon to which the user can drag files in order to encrypt or decrypt them. This patent is not particularly interesting or harmful but illustrates the level of detail and specificity the subject matter of a patent can have.
- European Patent 752695: Method and apparatus for simultaneously displaying graphics and video data on a computer display (Sun, 1997). This is a common technique for displaying information in a windowing system that has been in use for many years.
- European Patent 1043659: File signature check (Konami, 2000). This patent describes the use of checksums to detect whether files in a file system have been changed. This simple technique has already been used by many tools and is the obvious solution for this problem.
- European Patent 767940: Data pre-fetch for script-based multimedia systems (Intel, 2000). This patent aims at speeding up the execution of multimedia scripts running in a limited memory client. This is achieved by prefetching data references that occur in the script.
- European Patent 0195098: System for reproducing information in material objects at a point of sale location (Fpdc Inc, 1986).

  "This invention contemplates a system for reproducing information in material objects at a point of sale location wherein the information to be reproduced is provided at the point of sale location from a location remote with respect to the point of sale location, an owner authorization code is provided to the point of sale location in reponse to receiving a request code from the point of sale location requesting to reproducing predetermined information in a material object, and the predetermined information is reproduced in a material object at the point of sale location in response to receiving the owner authorization code."

[3] See, for instance, `http://www.base.com/software-patents/examples.html`.

[4] See `http://swpat.ffii.org/patents/samples/index.en.html` for a more extensive collection of trivial European patents.

This patent (on downloading and burning CDs!) was recently overturned by the UK High Court [28]. As the EPO did not overturn it, it is unclear whether the High Court would have been able to overturn this if the Directive that enforced the EPO's status quo to be uniformly implemented across the EU were in place.

To conclude. we mention some recent patent applications:

- European Patent 1046117: Web browser graphics management (Philips, filed 1999). This patent application describes a prefetching mechanism for web browsers that has been floating around for many years, for instance in the Mozilla browser, where it is called link prefetching.
- European Patent 1014627: Constrained shortest path routing method (Lucent, filed 1999). This application describes an algorithm for shortest path calculation and seems to be a variation on Dijkstra's algorithm [15].

As already stated, the above examples constitute cases where we suspect that triviality and/or existing prior art make these patents or patent applications undesirable. Clearly, a public effort should be launched to scrutinize the European patent data base and look for trivial software patents.[5] Another public effort that is badly needed is to set up a searchable archive of prior art for software.

8 Discussion

8.1 Expression versus idea

The common view on copyright versus patenting is that copyright protects the *expression* of an idea, while a patent protects the *idea itself*. One idea can be expressed in many different ways. In other words, copyright can protect one specific mystery novel, while a patent on the idea of a mystery novel itself will prevent anybody else to write mystery novels. The relevant US statute [36] reads as follows:

> *Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.*

U.S. courts have summarized this principle by stating that patents do not apply to ideas themselves, but to "implementations" of ideas, intending a broader and more inclusive sense of "implementations" than is commonly given in the software development community. It is akin to the notion that copyright protects expressions of ideas rather than ideas themselves [27]. A similar definition can be found in the TRIPS treaty [38] that regulates trade-related aspects of intellectual property rights:

> *... patents shall be available for any inventions, whether products or processes, in all fields of technology, provided that they are new, involve an inventive step and are capable of industrial application.*

Now it happens to be the case that software engineering has its own ideology about the distinction between idea and expression of that idea. A *specification* of a software system describes the desired functionality of a system and defines *what* is required. The specification leaves open many options *how* the specified system will be built (the *implementation*[6] of the system).

For a software engineer, it is hard to understand why software patents (supposed to be about ideas, e.g., specification level) end up having detailed flowcharts that belong to the implementation level. This raises the issue how software engineering and the patenting system interfere.

[5] The reader is invited to inspect the *Patent WIKI* database at `http://gauss.ffii.org/GaussFrontPage` and do some searching. It includes all patents issued by the European Patent Office. We can guarantee that there is some entertainment in this.

[6] The word "implementation" is a multi-faced sword that may easily cause harm. In a legal sense, as in the previous paragraph, implementation denotes any method to go from idea towards its realization. Following that meaning, the specification of a software system is already an implementation of the idea for that system. In a computer science context, implementation always refers to the actual building and realization of a system in software.

Another observation is that the legal language in patents is about software. Now it also happens to be the case that the topic of software specification has a long history in computer science and, from a computer science perspective, the legal descriptions in patents can in no way be classified as such. In that sense patent texts are technically (but, of course, not legally) unacceptable for software engineers, which has implications for their utility as disclosure documents.

In our opinion, the patenting literature should take good notice of what is known about describing software systems. A crucial observation is here that the notion of formally describing an idea does not occur in the software engineering literature and it will be very hard to achieve this in patent texts. Another observation is that we think that it is unavoidable that patent texts will become machine processable documents that will form an integral part of software in a similar fashion as specification, documentation, test cases and the like form an integral part of a software system.

A final, and also crucial, observation is that the requirement that a patent should make a "technical contribution" is hard to reconcile with software that lives in the realm of logical structures. In this way, software patents have to be expressed in unnatural ways that lead to under-protection as well as over-protection of certain inventions.

This is eloquently described by Plotkin [30], he proposes a reinterpretation of the patenting system from a software perspective. His observation is that there are crucial differences between the invention, description and patenting of electromechanical devices as compared to software programs. His key observation is that for electromechanical devices apart from a functional design, deriving a physical structural design is hard and also essential for obtaining patent protection. In the case of software, the logical structures described by the source code are the end point of human invention: the step to their physical realization is fully automated. Plotkin's objectives are the following:

> *A methodology is proposed for determining how particular areas of law should apply to software. The methodology asks and answers four questions: (1) What is software?, (2) How does software differ from other creative works?, (3) How are such differences legally relevant?, and (4) How should the law treat software in light of such differences? Application of the first half of this methodology reveals that computer programs have the unique quality of being human-readable and computer-executable instructions that describe actions in purely logical terms. Application of the second half of this methodology to patent law and the First Amendment to the US Constitution reveals that software's unique features violate the law's assumptions, leading to results that are at odds with the underlying public policies in each case.*

In later work [31] Plotkin proposes software patents in such a way that:

- *a software program be claimable solely in terms of its logical structure;*
- *a software program be patentable if:*
  - *the inventor provides a written description of the claimed logical structure;*
  - *the inventor provides a description that enables one of ordinary skill in the art to make and use the claimed program without undue experimentation;*
  - *the claimed logical structure has a practical utility;*
  - *the inventor conceives of the claimed logical structure;*
  - *the claimed logical structure is novel and nonobvious; and*
- *the scope of a software patent claim be limited to products and processes that embody the claimed logical structure.*

Other interesting proposals exist for reforming the patent system or for providing other forms of legal protection for software but they are not further discussed here. A concise summary of the history and current status of software patentability can be found in [20].

8.2 The role of patent claims

In [26] Lening and Cavicchi say about claims:

> *A claim is what an inventor is stating to be unique about the invention. The claims become the actual monopoly granted to the invention. Claims define the scope of protection granted to the invention.*

A claim can be *independent* (it stands by itself and is not dependent on another claim) or *dependent* (it makes express reference to a previous claim and depends on it). In the IsNot patent application, claim 1 is an independent claim while claim 2 is a dependent claim. A patent may also contain descriptions of preferred embodiments but they just serve as illustration and may at most be used to interpret the claims. A patent may contain both independent and dependent claims and the question arises what an infringement of a patent means *exactly*.[7] The precise procedure for interpreting the claims in a patent seems to be an "art" and is a matter of debate among lawyers [4]. This is unsatisfactory from a software engineering perspective. The patent is "infringed" (violated) if any one or more of the claims (independent or dependent) are infringed.

The status of claims needs also further clarification in the light of the "expression versus idea" discussion given earlier in Section 8.1. The question being: *what is an infringement?* From the perspective of the software engineering life cycle (Section 3) the following questions need clarification:

- Is infringement possible during requirements engineering?
- Is infringement possible during design?
- Is infringement possible during implementation?
- Is infringement possible during testing?
- Is infringement possible during maintenance?

To be on the safe side, we have assumed in our patent-based software life cycle that the answer to all these questions is "yes". However, the nature of such infringements will be completely different, both in their description, appearance, and discovery.

During requirements engineering and design, only the intended behavior of the system is available. It is for instance, impossible to observe a running version of the software. Infringements can only be discovered by a deep semantic comparison between patent text and design documents.

During implementation, the desired behavior is coded as software program. Now it becomes possible to observe the behavior of the software by executing it on a computer. It also becomes possible to perform more syntactic comparisons between patent text and program text.

Software is both human-readable and computer executable, and this makes it unique among patentable artefacts.

8.3 Software patent versus computer implemented invention

We have, so far, spoken about "software patents". However, the recently rejected EU patent directive speaks about "computer implemented inventions" (CII) rather than software patent.

It may be maintained that software patents do not exist and that CII is the right phrase to use. We completely agree that the notion of a computer implemented invention is a meaningful one and that such inventions may be in need of patenting. In such cases computer programs may be used as an implementation strategy but a pure hardware implementation may be conceivable as well. In our opinion, all patents discussed in Section 7 are software patents in a more generic sense. The patent is about how to achieve something by means of running computer executable programs (software), or even on methods for writing such programs or designing programming languages. None of these inventions makes any sense outside the realm of programmed computers, and these inventions are about how something may be achieved given

[7] The same questions can be asked when searching for prior art related to patent applications.

that computer programs will be used. *A software patent concerns an invention about a software-based computer implementation, while a computer implemented invention is about an invention that **may** be implemented in software.*

We cannot imagine that the IsNot patent application could be classified as a computer implemented invention which may admit a pure hardware embodiment, since this would amount to a single not gate. As a consequence the mere need to grant patents for clear cases of computer implemented inventions (e.g., the design of novel control software/hardware for an airbag) should *not* be taken as an argument that pure software patents do *not* exist. The software industry will soon be in a need to deal with a massive number of "true software patents". A careful consideration of the rules of that game from a software engineering perspective is necessary to grasp the effects of the introduction of patent regulations that will generate an abundance of such patents.

8.4 The role of "prior art"

Prior art is defined as the body of prior knowledge relating to the claimed invention, including prior use, publications and patent disclosures [26]. During the patent life cycle (Section 3), prior art plays a role at different moments:

- When an application is rejected, the applicant can dispute prior art that is used in the motivation of the rejection.
- When the patent is challenged, the challenger has to produce prior art that invalidates one or more of the claims of the patent.
- When the patent holder acts on an alleged infringement of its patent by a third party, he must show that the third party uses results or methods that are claimed by the patent (one could call this "posterior art").

In the patent application it is usually indicated which previous patents are used or extended. As already discussed in Section 5.2, only the European Patent Office does not require to mention prior art that is known to the applicant.

We conjecture that in all the three cases mentioned above, the determination of prior art is identical, whether this is true prior art or posterior art as defined above.

A patent may describe a technique that computer scientists consider to be trivial. Nonetheless, it may turn out to be very hard to find prior art for it. Well-known techniques cannot be published in a scientific publication for the simple reason that they are already well-known and do not constitute a new research result. These well-known techniques may be used in the source code of many software systems, but this does not count as "publication" and cannot be used to illustrate prior art. At the same time, it may also be the case that they are not covered by any patent and someone can just file a patent application for this well-known technique.

Ullman [35] is among the most cited computer science researchers world-wide and he describes eloquently the difficulty to find prior art for a patent application about matrix triangularization that was later used in an unsuccessful attempt to bring suit to the large spreadsheet manufacturers.

In disciplines like chemistry and biology the patent literature forms the actual documentation of inventions. For software the unique situation exists that there is another powerful information source that plays no role in the patent process: the source code itself. This is a major handicap when searching for prior art. There is evidence that cross-citation between the patenting literature and the computer science literature is nearly absent [1]. Compared to software patents, business patents seem to contain relatively more references to the non-patent literature [2]. Nonetheless, the world of software and the world of patents seem mostly disjoint. From this follows that computer scientists are currently not well-aware of the patent literature.

We may conclude that it is urgent to find new ways to establish prior art. One way is the creation of public web sites that solicit and award proofs of prior art. It seems reasonable to include procedures in the patenting system where the public can submit prior art against patent applications.

Another way for establishing prior art is the patent system itself. Suppose the IsNot application is rejected. This fact can have a very positive impact: all the claims in the application are considered to

be un-patentable and this blocks future patents on the issues stated in the rejected claims. In this way, a rejected patent application contributes to building up prior art. It is conceivable that major companies follow this strategy in order to prevent patent applications by competitors or to provide indemnity to clients against intellectual property claims.

8.5 Implications for Open Source Software

There has been active opposition from the Open Source Software (OSS) community against the emergence of a system for software patenting. As discussed earlier in Section 8.2, the assumption that open source software products allow inspection at a syntactic level does not imply a greater risk for infringement detection. To establish that this risk would be higher requires a very clear understanding of what constitutes an infringement of a software patent and how to establish such an infringement. As discussed earlier, exactly this understanding is missing. On the contrary, OSS producing companies or individuals may often afford to distribute quite vague functional specifications using the fact that their user community is willing to take some risks and to accept some trial and error, whereas producers of closed software components need to specify in meticulous detail what is to be expected from their products and this may even give better clues for those who search for potential patent infringements. We see therefore no reason why the authors of open source should be more (or less) worried about the potential implications of software patenting for their business than the authors of closed software, from the perspective of establishing infringement.

It is true, however, that the distributed development model of open source rests upon a legal infrastructure–open source licenses–that assume that individual authors own, and thus have the right to "give away", whatever they write. While this works in the copyright system, it is incompatible with patents, since individual developers can no longer assume that they own what they write, and can thus never know whether they have the right to "give it away" through open source licenses. This is a subject of further research in the course of the on-going study.

Many commercial manufacturers are now disclosing sources under limited licensing schemes while making use of substantial copyright protection. The variation of licensing schemes has much impact on the economic models used and only some licenses lead to the much debated cost reduction that many people consider typical for open source software. Source pricing and source disclosure are independent matters: open source software may even be quite expensive in some cases. If that were not the case (in principle) the whole patenting system should be considered irrelevant as such because it only protects users and producers of disclosed information.

8.6 Implications for education

Patent law requires that a patent should be non-obvious to a "person of ordinary skill in the art". Note that this skill regards a *technical* art and that the person is not expected to have any legal knowledge or ability to interpret the legal meaning of a software patent. As already observed by Ullman [35], it is unclear what the technical background of such a person should be: ranging from a self-educated programmer, via a bachelor or master in computer science or software engineering, to a professional researcher in these areas. If we consider the Software Engineering Body of Knowledge (SWEBOK [34]) as approved by IEEE, we are pretty sure that a person with that technical knowledge is unable to read or interpret software patents let alone determine potential infringements. The patent-aware software engineering life cycle (Section 3) also requires an increased level of awareness of software patents among software engineers as well as the skills to turn this awareness into deeds.

It is clear that the current education of software engineers and the future requirements imposed by a patenting system including software patents will be dramatic. As far as we aware, there is no curriculum worldwide that is prepared for this. Governments should invest in the development of such curricula and in major retraining of professional software engineers.

8.7 Implications for government-funded research

Software is developed in many research projects that are being funded by national governments. Most of these project follow a traditional software life cycle that ignores patents. In order to avoid that governments

become vulnerable for extensive infringement claims, they should require that these projects switch to at least a patent-aware software life cycle. This will require extensive additional funding for these projects.

8.8 Implications for the debate on the software patents

The introduction of software patents in any form immediately raises the following questions:

a What constitutes prior art, and what is the status of existing programs.

b How to avoid trivial patents.

c How to design a patent-based software engineering life cycle.

d How to design a patent aware life cycle (less crucial but economically vital).

In the recent debate in the EU we get the impression that the (recently rejected) directive "on the patentability of computer-implemented inventions" (software patent directive), as it stands would lead to de-facto software patents (in spite of the CII jargon) without the prerequisite clarification concerning the issues listed above, thus creating unpredictable legal risks for many parties involved.

Amending the directive to such extent that there is no legal basis left for the protection of any software components deprives manufacturing companies from legal means available to them now, and thereby introduces additional risks just as well. This seems to lead to the position that neither the directive nor a version of it that cuts out any IPR protection for software components (or against infringing software components) is a step forward.

The rejected proposal seemed to focus on software/hardware component specifications that constitute a vital part of CII's. The functional specification of a unit is given (as part of the proposed CII architecture) and then an infringement may result by producing a software component that meets the specification even if the manufacturer has shown the ability to implement the specification by means of the description of a piece of hardware. Thus some branches of industry propose (understandably) a capability to provide this form of protection. Unfortunately, the resulting scope of IPR and infringement protection has been insufficiently demarcated.

The modularization paradox Some assume that by requiring that embodiments of a patent have effects that depend on laws of nature (though excluding software as such) conceptual problems can be solved. This cannot be excluded *per se* though it may get paradoxical as follows:

- One may describe an invention as an application in technology rather than dealing with laws of nature.
- One may consider computer programs as software and one may also consider software as belonging to technology.
- Now consider computer programs P and Q where Q provides a context within which P may work. On the one hand $P + Q$ cannot be patented as it is 'software as such', on the other hand P may be patented because of its role it may play in the context of Q (which is a technical context given the above assumptions).

Taking this observation to the extreme: in a context where software as such cannot be patented and technical effects are required, one may be tempted to split a software invention into claimed components and stated components where the stated components are part of the justification of the claimed components. Interestingly this introduces a tendency to trivialize a patent description. More importantly, however, the whole state of affairs with P and $P+Q$ is conceptually inconsistent. Therefore the dogma's that software as such cannot be patented *and* technical effects are required make sense only in a setting where one assumes beforehand that a collection of software components never represents a part of technology.

How to move ahead? Given the fact that world-wide a large number of de-facto software patents exist (even if a jargon is used that suggests these patents to be of another nature) it is already now important for the EU to initiate substantial research and development for the clarification of the questions **a**–**d** mentioned above. On the basis of such work technology can be developed that takes into account all existing patent databases. In successive stages limited possibilities for the protection of

- software/hardware components specifications,
- software component implementations,
- software architectures,
- software processes (software engineering methods)

may be developed.

By doing this kind of work the EU will possibly lead the way in sophisticated use of software patent databases while at the same time preparing for patenting regulations that really work. In terms of software engineering these regulations themselves are just some form of standard concerning the software process. It is clear that such a standard should only be enforced if it has substantial informal backing and if the technology supporting it is sufficiently sophisticated.

We expect that in the long run software patents will indeed emerge and that this will lead to a wealth of supporting technology. What is at stake here, is the risk that the EU misses the opportunity to leap ahead by developing sophisticated legislation in which software is a first class citizen and *also* misses the opportunity to develop the technology for supporting such legislation.

That leads us to this position: a sophisticated patenting system for (categories of) computer software will enhance software technology in the EU, provided that the considerations given here are addressed on a reasonably grand and effective scale. Introduction of a software patenting system without these prerequisites in place will have disappointing effects. The opposition against the proposed directive as well as its rejection in the parliament are a manifestation of these disappointing effects.

8.9 Integrity axiom for software patent authors and owners

We recommend to add the following integrity axiom to the assumptions about software patents: *every patent which is either live or in the application phase expresses the views held by its authors and owners in the following way*:

- The described invention did not conflict with prior art (in the most general sense of this expression) when it came into existence and by definition has been so ever since.[8] In addition, the patent is non-trivial at that same moment in time.
- The patent authors rightly claim as professional software engineers the IPR for said invention.
- This IPR entitles them to economic revenues in an enforcible way.
- If the patent is owned by an organization that employs one or more of its authors, the relevant management layers of this organization share the views stated above.

This axiom is non-obvious because filing a patent application is an action by some agent and the axiom is about the mental state of that agent.

One might drop the integrity axiom in which case software patenting becomes some form of gaming not primarily based on the meaning of the patents but rather on their tactical and dynamic properties. For instance, a company might file a sequence of trivial patents just to exhaust the capacity of an economic opponent to effectively complain about these applications in order to arrive at a stage where IPR can be claimed even if it is not justified in real terms. But if the only way to get something out of patents would be along these lines we tend to agree with Knuth [25] that the whole enterprise is flawed. The integrity axiom excludes tactical patenting which is not based on reliable facts. This is very similar to scientific publications which are also supposed to adequately represent author's views.

[8] US Patent Law requires a declaration from each applicant stating that "he believes himself to be the original and first inventor of the [technology] for which he solicits a patent."

8.10 A research agenda for software patent research

Taking software patents seriously means designing patenting systems and studying their implications. Here are some suggestions for a research agenda.

Current status of software patenting regimes One should take into account at least what happens in the USA, the EU, Japan, India and probably more. The Gauss database mentioned earlier is an example of such work. Here one finds the systematic investigation into the non-triviality and prior art violations of existing patents. We can imagine that a patent monitor is developed which enables the public to systematically submit their opinions about existing patents. In addition various forms of text-mining and cluster analysis can be employed to unlock the knowledge in the patent databases.

Revision proposals concerning the various regimes Several proposals have already been made for revising the various regimes. These should be studied and compared in detail.

Designing possible software patenting regimes There is no reason to believe that one unique software patenting regime can be designed, assuming that one exists at all. Thus many different regimes should be investigated. For each regime a set of questions has to be settled: what constitutes prior art, what is an infringement, how to define the particular 'patent speak' and its semantics, definition of the appropriate life cycles, and so on.

An important step might be to develop a collection op hypothetical software patents, i.e., rewrites and perhaps simplifications of the software development history in which known developments are ordered in such a way that some steps can convincingly be patented. The historical development of computer software might even be simulated in a game-like fashion in order to study the impact that some patents (had they existed) might have had.

Collection of prior art A crucial element in any patenting regime is the role of prior art. We propose to investigate the possibilities for

- Formalizing prior art, i.e., all relevant knowledge about software. This would include patents, (non-) academic publications, and any other relevant information.
- Formalizing the claims in patents, although it is unclear to what extent this is possible or even desirable. Although ambiguity in patent claims can be harmful, some amount of ambiguity should be tolerated. One role for judges is interpreting these ambiguities in light of technological developments that occurred after the claims were written, and other changed circumstances. It would be difficult if not impossible to retain such ambiguity if patent claims were formalized.
- Comparing formalized patent claims with formalized prior art.
- Automated searches for patent infringements in existing software, given formalized patent claims.

We believe that this research agenda can contribute to a revision of the patent system and may even lead to a form of software patents that behave as intended: disseminate the knowledge about inventions and give rewards to true inventors.

9 Conclusions, policy suggestions and further research

Our main conclusion is that patents are too important to be left to lawyers and economists and that the only way to fully understand the ramifications of software patents on existing software engineering is to completely reinterpret the patenting system from a software engineering perspective. This will require extensive study and will also create competitive advantages for the EU.

9.1 Conclusions

1. Software is both human-readable and computer executable and this makes it unique among patentable artefacts. The requirement that a patentable invention should make "a technical contribution" leads to unnatural descriptions of software inventions and to inadequate claims.

2. There is a need for a patent life cycle that can be used to better understand the patenting process; in this paper we propose such a life cycle. Since software developers work worldwide, the patent life cycle should abstract from specific patenting regimes (EU, US, Japan).

3. We propose software life cycles that are patent-aware (defensive), patent-based (offensive), and IPR-based (includes copyright, patents and secrecy). They are needed to reconcile software engineering practices with the patenting system.

4. Adopting any patent-related software life cycle increases the costs of software development.

5. The fact that patenting of certain computer implemented inventions might be reasonable should be considered independently from the implications of pure software patents. New forms for the protection of software inventions should be studied.

9.2 Policy suggestions

We come to the following policy suggestions based on the analysis given in this paper:

1. The European Patent Office should require that patent applications mention all prior art (not only from the patent literature but especially from sources outside the patent literature) that is known to the applicants. In practice, disclosure or even awareness of prior art is avoided for legal reasons (see the discussion on "Chinese walls" in Section 3). This is an undesirable situation since it undermines one of the primary roles of the patenting system: acting as a knowledge dissemination mechanism.

2. A public effort should be launched to scrutinize ("garbage collect") the European patent data base and look for trivial software patents. Such a public validation phase should become part of the patent application procedure.

3. The sources on which prior art searches are based should be extended in the case of software patents; in particular web-sites, mailing lists, and software source code should be permitted as sources of prior art.

4. Governments should make major investments in designing patent-based curricula for software engineering and computer science as well as in retraining programs for professional software engineers.

5. Governments should require that all software development that takes places in projects they fund follow the patent-aware software life cycle. Otherwise, governments may become vulnerable for infringement claims.

6. Rejected trivial software patents are a tool for establishing prior art. The EU should launch collaborative efforts to collect and categorize prior art in software engineering. This will lead to a defense against software patents from outside the EU and it will also advance the level of knowledge and technology to effectively handle patent information.

9.3 Further research

The IPR-based software life cycle already raises many fundamental questions that are not easy to answer:

- Is it possible to use these extended software life cycles in such a way that they comply with the major patenting systems world wide?

- How can the software engineering knowledge that is hidden in the patent data bases made accessible for software engineers?

- Is it possible to a give an *operational* definition of a patent infringement that can be used by software engineers?
- For each of the phases of the software life cycle (requirements engineering, design, implementation, testing and maintenance) the following questions should be answered:
  - How is knowledge in this phase represented?
  - Where can prior art for this phase be found?
  - How can patent infringements in this phase be identified?
  - How can patent infringements in this phase be resolved?

  We expect that the answers to these questions will widely differ for each phase.
- What are the technical implications for software development when using these extended software life cycles?
- What are the economic implications of the extended software life cycles?

Taking software patents seriously means designing patenting systems and studying their implications for open source software as well as for open standards. We believe that studying the following research questions can contribute to a revision of the patent system and may even lead to a form of software patents that behave as intended: disseminate the knowledge about inventions and give rewards to true inventors. We propose therefore to study the following research questions, partly in the context of the current project:

- *Knowledge Extraction from Current Patent Databases:* how can the software engineering knowledge that is hidden in the current patent databases be unlocked and be made available to the software engineering community?
- *Prior Art:* how can knowledge about prior art in the domain of software engineering be described and formalized in such a way that trivial patents can be avoided?
- *New Patent Systems:* Design a patent system with the following properties: (a) Patent applications are written as a formal description of the claimed inventions; (b) Applications can be compared automatically with the patent database; (c) Patents enable the automatic detection of infringements.

We expect as results from studying these questions:

- An approach for knowledge mining in patent databases.
- An approach to formal representation of software engineering knowledge.
- Techniques for the formal description of patents.
- Ideas for the design of new patent systems.
- Insights in the implications of software patents for open source software and open standards.

Research Method The Gauss database[9] is an example of work aiming at the dissemination of knowledge about software patents.

Another public effort that is badly needed is to set up a searchable archive of prior art for software. Here one finds the systematic investigation into the non-triviality and prior art violations of existing patents. It is desirable to develop a patent monitor which enables the public to systematically submit their opinions about existing patents. In addition various forms of text-mining and cluster analysis can be employed to unlock the knowledge in the patent databases.

A crucial element in any patenting regime is the role of prior art. We propose to investigate the possibilities for

[9]The reader is invited to inspect the *Patent WIKI* database at `http://gauss.ffii.org/GaussFrontPage` and do some searching. It includes all patents issued by the European Patent Office. We can guarantee that there is some entertainment in this.

- formalizing prior art, i.e., all relevant knowledge about software;
- formalizing the claims in patents;
- comparing formalized patent claims with formalized prior art;
- automated searches for patent infringements in existing software, given formalized patent claims.

Several proposals have already been made for revising the various regimes. These should be studied and compared in detail.

There is no reason to believe that one unique software patenting regime can be designed, assuming that one exists at all. Thus many different regimes should be investigated. For each regime a set of questions has to be settled: what constitutes prior art, what is an infringement, how to define the particular 'patent speak' and its semantics, definition of the appropriate life cycles, and so on.

An important step might be to develop a collection op hypothetical software patents, i.e., rewrites and perhaps simplifications of the software development history in which known developments are ordered in such a way that some steps can convincingly be patented. The historical development of computer software might even be simulated in a game-like fashion in order to study the impact that some patents (had they existed) might have had.

Related Research Relevant are studies on legal and economic effects of patents. For economic effects we refer to [21, 11] and for legal aspects to [32, 5].

A survey of the state of the practice of software patenting can be found in [17] and [26] gives a useful glossary for patent searches. A Software Engineering Body of Knowledge can be found in [34].

A crucial observation is that the requirement that a patent should make a "technical contribution" is hard to reconcile with software that lives in the realm of logical structures. In this way, software patents have to be expressed in unnatural ways that lead to under-protection as well as over-protection of certain inventions.

This is eloquently described by Plotkin [30], he proposes a complete reinterpretation of the patenting system from a software perspective. His observation is that there are crucial differences between the invention, description and patenting of electromechanical devices as compared to software programs. His key observation is that for electromechanical devices apart from a functional design, deriving a physical structural design is hard and also essential for obtaining patent protection. In the case of software, the logical structures described by the source code are the end point of human invention: the step to their physical realization is fully automated. In [31] this view is further elaborated.

Other interesting proposals exist for reforming the patent system or for providing other forms of legal protection for software but, for reasons of brevity, they cannot be further discussed here. A concise summary of the history and current status of software patentability can be found in [20]. A proposal for a "third paradigm" between copyright and patents can be found in [18].

Although we have mostly argued for the elimination of trivial patents and do not draw the conclusion that software patents are a bad idea under all circumstances, we cannot resist to conclude this paper with a quote from the world-famous Donald Knuth, professor emeritus from Stanford University, in a letter to the US Patent Office [25]:

> *The basic algorithmic ideas that people are now rushing to patent are so fundamental, the result threatens to be like what would happen if we allowed authors to have patents on individual words and concepts. Novelists or journalists would be unable to write stories unless their publishers had permission from the owners of the words. Algorithms are exactly as basic to software as words are to writers, because they are the fundamental building blocks needed to make interesting products. What would happen if individual lawyers could patent their methods of defense, or if Supreme Court justices could patent their precedents?*

Acknowledgments

We thank our partners in the *Study of the effects of allowing patent claims for computer-implemented inventions* for their insights and help. Reinier Bakels was very helpful in answering our questions about legal matters, Bronwyn Hall pointed to relevant references, and Rishab Ghosh kept pressuring us to include more European patent examples. All three reviewed drafts of this paper. We also thank the EU reviewers Alfonso Fuggetta and Manuel Martínez Ribas for their comments.

Robert Plotkin made excellent comments (which we sometimes used literally in the final version of this paper) and we also thank him for our ongoing discussion on the topic of IPR. Erik Josefsson was helpful in pointing us to interesting EU patent applications and for setting up `http://gauss.ffii.org/GaussFrontPage` as a useful tool for patent research. Dirk-Willem van Gulik draw our attention to the importance of Chinese walls between software developers and patent attorneys. Paul E. Merrell pointed to inaccuracies in our draft descriptions of what is patentable and his suggestions greatly helped to clarify this. Jan van Eijck helped to increase our insight by challenging our assumptions about patents and Jo Lahaye was a stimulating discussion partner on this topic. Finally, we thank the EU project officers Enrica Chiozza and David Callahan for their continued support for this project.

References

[1] G. Aharonian. Patent examination system is intellectually corrupt. `http://www.bustpatents.com/corrupt.htm`, May 2000.

[2] J.R. Allison and E. H. Tiller. Internet business method patents. In W. M. Cohen and S. A. Merrill, editors, *Patents in the Knowledge-Based Economy*, pages 259–284. National Research Council, Washington, National Academies Press, 2003.

[3] ANSI. `http://www.fortran.com/F77_std/f77_std.html`, 1977. ANSI Standard X3.9-1978 and ISO 1539-1980.

[4] R. Bakels, 2005. Private Communication.

[5] R. Bakels and P.B. Hugenholtz. The patentability of computer programmes: Discussion of European level legislation in the field of patents for software. Technical report, European Parliament, 2002.

[6] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[7] J.A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.

[8] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[9] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of logic and algebraic programming*, 51(2):125–156, 2002.

[10] J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische software engineering, kenmerken-faseringen-classificaties*, volume XXVIII of *Questiones Infinitae*. Zeno instituut voor Filosofie (Leiden-Utrecht), 1998. (In Dutch).

[11] J. Bessen and R.M. Hunt. An empirical look at software patents. Economics Research Working Paper 03-17/R, Philadelphia Federal Reserve Bank, March 2004.

[12] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[13] Software patents: the choice is yours. http://www.softwarepatenten.be/conferenties/september03, September 17, 2003. Brussels.

[14] Digital Equipment Corporation. *Processor Handbook PDP11/45*, 1974.

[15] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[16] ECMA International. *C# Language Specification*, 2nd edition, December 2002. ECMA-334, http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[17] Foundation for a Free Information Infrastructure (FFII). Software patents: Questions, analyses, proposals. http://swpat.ffii.org/analysis/index.en.html/, Visited August 28, 2005.

[18] Foundation for a Free Information Infrastructure (FFII. Third paradigm between patent and copyright law. http://swpat.ffii.org/analysis/suigen/index.en.html, Visited August 28, 2005.

[19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[20] J. Halbersztadt. Remarks on the patentability of computer software – History, Status, Developments. http://swpat.ffii.org/events/2001/linuxtag/jh/swplxtg017jh.en.pdf, April 2001.

[21] B. H. Hall. Innovation and market value. In R. Barrell, G. Mason, and M. O'Mahoney, editors, *Productivity, Innovation and Economic Performance*, pages 177–198. Cambridge University Press, 2000.

[22] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[23] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[24] P. Klint. Een patentoplossing? Nee, dank U! *I/O Informaticaonderzoek*, 1(2):3, September 2004. (In Dutch), http://www.informaticaplatform.nl/images/uploaded/magazine_2004_12_IO2t%otaal.pdf.

[25] D. Knuth. Letter to the US patent office. http://lpf.ai.mit.edu/Patents/knuth-to-pto.txt, September 2003.

[26] C. Lening and J.R. Cavicchi. Patent searching glossary. Technical report, Franklin Pierce Law Centre, 2003. http://ipmall.info/hosted_resources/patent_searching_glossary.pdf.

[27] P.E. Merrell, 2005. Private Communication.

[28] M. Murphy. Getty and corbis win image patent dispute. Seattle Post-Intelligencer http://seattlepi.nwsource.com/business/227728_gettycorbis09.html, June 2005.

[29] J. Park. Evolution of industry knowledge in the public domain: Prior art searching for software patents. *SCRIPT-ed*, 2(1), 2005. http://www.law.ed.ac.uk/ahrb/script-ed/vol2-1/park.asp.

[30] R. Plotkin. From idea to action: toward a unified theory of software and the law. *International Review of Law, Computers & Technology*, 17(3), November 2003.

[31] R. Plotkin. Computer programming and the automation of invention: a case for software patent reform. Working Paper Series, Public Law & Legal Theory Working Paper 04-16, Boston University School of Law, 2004.

[32] P. Samuelson. Should program algorithms be patented? *Communications of the ACM*, 33(8):23–27, 1990.

[33] 30 European Computer Scientists. Petition to the european parliament on the proposal for a directive on the patentability of computer-implemented inventions. *CEPIS UPGRADE The European Journal for the Information Professional*, IV(3):24–25, June 2003. http://www.upgrade-cepis.org/issues/2003/3/upgrade-vIV-3.html.

[34] Software engineering body of knowledge (SWEBOK). http://www.swebok.org, 2004.

[35] J.D. Ullman. Ordinary skill in the art. http://www-db.stanford.edu/~ullman/pub/focs00.html, November 2000.

[36] United States Code (USC). Title 35, Section 101: Inventions patentable. http://caselaw.lp.findlaw.com/scripts/ts_search.pl?title=35&sec=101, January 22 2002.

[37] H. van Vliet. *Software Engineering: Principles and Practice*. Wiley, second edition, 2000.

[38] World Trade Organization (WTO). Trips: Agreement on trade-related aspects of intellectual propery rights, Section 5, Article 27: Patentable Subject Matter. http://www.wto.org/english/tratop_e/trips_e/t_agm3c_e.htm.