Efficiently Retrieving Function Dependencies in the Linux Kernel Using XSB

Spyros Hadjichristodoulou, Donald E. Porter, and David S. Warren

Computer Science Department
Stony Brook University
Stony Brook, NY 11794-4400
{shadjichrist,porter,warren}@cs.stonybrook.edu

Abstract. In this paper we investigate XSB-Prolog as a static analysis engine for data represented by medium-sized graphs. We use XSB-Prolog to automatically identify function dependencies in the Linux Kernel—queries that are difficult to implement efficiently in a commodity database and that developers often have to identify manually. This project illustrates that Prolog systems are ideal for building tools for use in other disciplines that require sophisticated inferences, because Prolog is both declarative and can efficiently implement complex problem specifications through tabling and indexing.

Keywords: Linux Kernel, Function Dependencies, XSB

1 Introduction

When the idea of Logic Programming was conceived in the early 1970s [1], its primary application was Natural Language Processing (NLP) [2]. The purely declarative nature of Prolog programs allows programmers to specify a problem's requirements, and then leave the Prolog engine to actually solve it. Over the past years, researchers have developed a range of Prolog optimizations, including indexing [3] and incremental tabling [4,5], which make Prolog-generated solutions efficient. Because programmers can easily specify solutions in Prolog, and because the generated solutions are efficient, it has the potential to be a practical tool for general-purpose problem solving. However, Prolog has not been widely adopted outside Artificial Intelligence (AI); it is traditionally employed for applications in expert system building [6], ontology representation [7,8] and theorem proving [9]. One of the most notable and recent example of an application of Prolog in NLP is the implementation of IBM's Watson computer system [10]. In 2011, Watson competed against former American television "Jeopardy" game winners in answering questions posed in natural language and won the grand prize[1].

---

[1] http://www.computerworld.com/s/article/9209938/Watson_triumphs_in_em_Jeopardy_em_s_man_vs._machine_challenge

Logic Programming has been adopted by industry as well, primarily to make complex inferences over a data set [11,12,13,14], or as a security policy specification language [15]. Despite these advances showing the utility of Logic Programming for developing robust software systems, many opportunities for further adoption remain. We believe that Prolog's combination of easy-to-specify solutions with the efficient implementations makes Prolog ideal for use in other CS disciplines as well, especially where analysis, knowledge representation, and inference over large amounts of data is needed.

One example of a CS discipline where such analyses are needed is Operating Systems (OS). Operating systems are usually written to support a wide range of hardware, including different instruction set architectures, and a single OS is often several million lines of code written in multiple languages. For instance, the Linux Kernel has components written in C and various assembly dialects, and its total code base exceeds 15,000,000 lines of code. What makes understanding the Linux Kernel even more difficult is its complex set of compilation options, which are in turn implemented by heavy use of C preprocessor macros and, in some cases, multiple versions of the same function—all of which can obfuscate the code and frustrate simple text searches.

There are many development tasks that require an expert to understand and manually reason about this large body of code. For instance, certain synchronization primitives in Linux, such as spinlocks and read-copy update (RCU), require that a blocking function not be called while in a critical section. In order to add a function call inside a critical section without violating this invariant, one must essentially determine whether `schedule()` is in the transitive closure of all functions that could be called by a newly-added line of code.

The Linux Cross-Reference (LXR[2]) is a tool that helps developers understand the Linux kernel source. Because the LXR is implemented using a traditional RDBMS, PostgreSQL in particular, the RDBMS can efficiently execute simple queries, such as locating all instances of a particular string in the code. However, a traditional RDBMS does not offer any kind of *reasoning* or *inference* over the data. In our example of finding whether `schedule()` is in the transitive closure of a function call, answering this query in a traditional RDBMS would require loading all of the tables in memory, and then joining the tables using the callee function's name as the key. This would be too inefficient for databases containing information about huge code bases such as the Linux Kernel. Further, handling the case where function `A` calls a chain of intermediate functions that ultimately call function `B` further increases these overheads in a traditional RDBMS.

Queries that require such inferences or deductions can be implemented using a *deductive database*—a database optimized for deductions over large data sets. XSB-Prolog is among the most efficient deductive database systems available [16], hence it is ideal for inferring information from medium and large datasets.

This paper describes the design and preliminary experiences using XSB-Prolog to build a tool to help developers reason about the Linux kernel source

---

[2] http://lxr.linux.no/

code. This paper focuses on the transitive closure problem described above; we are extending the tool as ongoing work. Our tool is available at http://ewl.cewit.stonybrook.edu/spyros/kernel.php. Section 7 presents space and time measurements, demonstrating that tabling and indexing in Prolog systems make the difference between practical and impractical tools.

2   Description of the Problem

As any large code base evolves, developers may discover the need to modularize and reuse functionality. For instance, file systems often "reinvent the wheel" in developing similar API features or techniques for managing consistency across metadata writes. Linux supports dozens of file systems that offer various features and performance characteristics; although some components are shared (e.g., the Linux libfs and the ext3 jbd journal), separating a feature into a reusable module is a manual process only undertaken by an expert [17]. As a result, once a feature has proved useful in one file system, the feature is not easily adapted to all other file systems. Thus, useful features languish in individual file systems or research prototypes, such as transactions [18,19], atomic append [20] and copy-on-write checkpointing [21].

A key question a developer must answer when modularizing code is essentially: where is the most functionally narrow point in this code base at which to create a shared API? Or, for any given line of code or function call used in the implementation of a feature, does it make more sense to bring along supporting code? As another example of this issue, consider porting a data structure from a user-level library into the OS kernel: for each library call the data structure makes, should the developer copy in that library function, adapt to a similar function provided in the kernel, or re-engineer that part of the code to avoid the use of the library call? These design decisions can be subtle, and the designer could benefit greatly from a tool that automatically identifies how difficult a given function is to excise from its supporting code base.

Finally, even more mundane tasks require OS kernel developers to reason about the transitive closure of all possible functions a given function could call. For instance, if a developer is modifying a function that acquires a lock, the developer must not call any functions that could acquire a second lock that violates the kernel's global lock ordering—requiring the developer to understand all possible code paths or risk introducing deadlocks. Similarly, read-copy update (RCU) [22] is designed with the invariant that a reader will not call a blocking function inside a critical section; this again requires a deep understanding about all possible disk reads, network accesses, memory allocations, etc. Although Linux can compile in dynamic checks that can catch these bugs, these tools will only work if the checks are correctly written and all code paths are tested. Developers could avoid ever introducing these subtle bugs if they had the ability to double-check these global invariants while writing the code.

3   Our Solution

A key observation of this work is that the power of a tool to help users make inferences about a large dataset is determined by the power of the underlying DBMS. Because LXR is built using a traditional RDBMS, it cannot support even simple queries that require recursion. Trying to approximate recursive queries in PostgreSQL would require multiple joins of large tables. In XSB-Prolog however, solutions to such problems are both easy to specify and efficient, because tabling ensures termination and efficiency of the various transitive closure definitions (left-recursive, right-recursive, and doubly-recursive). Our aim is to facilitate issuing simple queries with sophisticated implementations by OS developers and researchers. We further aim to encapsulate the details of how Prolog works and how these queries are implemented from the users. Even if a sophisticated engine is required to answer these queries, the user interface should be simple and intuitive.

Thus, the following key components are required for the development of our tool, as explained in the following sections:

1. Extracting function dependency information from the Linux Kernel.
2. Representing the function dependencies in a way that is easily processed by Prolog systems.
3. Designing an easy-to-use interface for this tool, accessible even for users who are not proficient in Prolog.

4   Extracting Function Dependency Information from the Linux Kernel

In order to extract function dependency information from the Linux Kernel, we used `GCC` and `egypt`[3]. The `egypt` tool is a Perl script that parses the intermediate code representation of C source files and outputs a relevant `Graphviz`[4] file, which can be used to graphically represent the call graph.

The `egypt` tool takes the compiler's intermediate code representation as input; to output this intermediate representation, we compile the source code using the `GCC` using the `fdump-rtl-expand` compilation flag. The only change to the Linux kernel is adding this flag to the makefile. By compiling the entire kernel with this extra flag, we get one extra output file per C source file with extension `.c.144r.expand`. These files contain intermediate code information in the form of *Register Transfer Language* (RTL).

Running `egypt` on each of these RTL files outputs call-graph information in an easy-to-read manner, i.e. in the form of `Function A -> Function B` expressions. The Linux Kernel version we used is **3.6.6**, and we performed an `allnoconfig` compilation, which is a minimal configuration that disables all optional features.

---

[3] http://www.gson.org/egypt/egypt.html
[4] http://www.graphviz.org/

```
#ifdef CONFIG_STACK_GROWSUP
int expand_stack(struct vm_area_struct *vma, unsigned long address)
{
        struct vm_area_struct *next;

        address &= PAGE_MASK;
        next = vma->vm_next;
        if (next && next->vm_start == address + PAGE_SIZE) {
                if (!(next->vm_flags & VM_GROWSUP))
                        return -ENOMEM;
        }
        return expand_upwards(vma, address);
}
#else
int expand_stack(struct vm_area_struct *vma, unsigned long address)
{
        struct vm_area_struct *prev;

        address &= PAGE_MASK;
        prev = vma->vm_prev;
        if (prev && prev->vm_end == address) {
                if (!(prev->vm_flags & VM_GROWSDOWN))
                        return -ENOMEM;
        }
        return expand_downwards(vma, address);
}
#endif
```

Fig. 1. An excerpt from `mm/mmap.c` in Linux 3.10, illustrating the use of the C preprocessor to select between two different implementations of a function.

One drawback of our current design is that, because we extract dependencies after the preprocessor runs, we cannot easily capture function dependencies that might arise in a different configuration. Recall that the Linux Kernel provides many compilation options, which are generally selected using C preprocessor macros. For example, whether a program stack adds frames at a higher or lower virtual address (grows "up" or "down") is controlled by a compile-time option. Figure 1 lists an excerpt of `mm/mmap.c` from Linux 3.10, which illustrates the potential to miss possible dependencies.

Because our current prototype identifies function dependencies after `CONFIG_STACK_GROWSUP` is evaluated, it can either miss `expand_upwards` or `expand_downwards` as a potential dependency. We are currently investigating ways to retain the simplicity of RTL without losing information about the preprocessor configuration directives.

5   Easy-to-process Representation of Information

This section describes how we process the output of `egypt` using XSB-Prolog, and then use this information to assert facts describing the call graph into the database. The output `.eg` files generated use a fairly small subset of the `Graphviz` language, specified by the grammar below:

$\langle$graph$\rangle$ ::= "**digraph**" "**callgraph**" "{" $\langle$graph_descr$\rangle$ "}"

$\langle$graph_descr$\rangle$ ::= $\langle$id$\rangle$ ";" $\langle$graph_descr$\rangle$
          | $\langle$id$\rangle$ "−" ">" $\langle$id$\rangle$
          | $\langle$style_descr$\rangle$ $\langle$graph_descr$\rangle$
          | "ϵ"

$\langle$style_descr$\rangle$ ::= "[" "**style**" "=" $\langle$style$\rangle$ "]" ";"

$\langle$style$\rangle$ ::= "**solid**" | "**dotted**"

$\langle$id$\rangle$ ::= """ $\langle$underscores$\rangle$ $\langle$ident$\rangle$ """

$\langle$underscores$\rangle$ ::= "_" $\langle$underscores$\rangle$ | "ϵ"

The *ident* identifiers are tokens containing English characters and possibly integers. We parse this representation of the call-graph information using XSB's *DCG*s (Definite Clause Grammars). We also implemented a tokenizer using the generic scanner in XSB-Prolog for recognizing Modula-3 and Java programs[5]. This scanner splits the input in a list of token positions, and passes that information to the parser. As we use the grammar to parse each edge represented in the `.eg` file, we use the representation encoded to assert `edge/2` facts for each edge. These facts have the format `edge(File1':'Source,File2':'Dest)`.

Within the same source file, the variables `File1` and `File2` will be bound to the same atom, which is the name of the source file. However, when all these edges are asserted, it is useful to distinguish between different file names when we implement the transitive closure. Our parser is a direct translation of the above grammar, and can be found below:

```
graph(File) --> ['digraph'], ['callgraph'], ['{'], graph_descr(File),
  ['}'].

graph_descr(File) --> identifier(_), [';'], graph_descr(File)
        | identifier(Source), ['-'], ['>'], identifier(Dest),
        { assert(edge(File':'Source,File':'Dest)) },
        style_descr, graph_descr(File)
        | [].

style_descr --> ['['], ['style'], ['='], style, [']'], [';'].

style --> ['solid']
        | ['dotted'].
```

---

[5] Available in www.cs.stonybrook.edu/~shadjichrist/scanner.P

```
identifier(Id) --> ['"'], underscores, [ident(Id)], ['"'].

underscores --> ['_'], underscores
        | [].
```

After each `.eg` file is processed, appropriate `edge/2` facts will have been asserted into memory, giving us a representation of the graph of the transitive closure, which Prolog can easily process. The `parse_files/0` predicate handles parsing the `.eg` files; the `edge/2` assertion happens during parsing, as shown in the above code (`scan_file/2` is implemented in the scanner library mentioned earlier).

Since we are asserting information gathered from the entire Linux Kernel, one can imagine that the number of `edge/2` facts that lies in memory is quite large, so it is useful to check how much time it takes to assert all these data into memory. The predicate `qtime/2` calculates the time a given query needs to be computed, so the appropriate call below gives the time needed:

```
|?- qtime(parse_files,T).
T = 104.2960
```

Perhaps unsurprisingly, it takes almost 2 minutes to assert all the edges into memory. To see how many of these facts are being asserted, a call to `findall/3` can be used:

```
|?- findall(_,edge(_,_),L),length(L,N).
N = 52955;
```

We see that approximately 53,000 edges are being asserted into memory. Although the assertion time may appear high at first glance, XSB-Prolog offers the solution to such problems by the means of advanced *indexing* techniques, including *trie indexing* [16]. By adding an `:- index(edge/2,trie).` directive, an index is created for the `edge/2` facts which are now asserted much faster, as `qtime/2` divulges:

```
|?- qtime(parse_files,T).
T = 19.6910
```

As a result of using indexing, we have reduced the time of needed to assert the data-to-process in memory by a factor of more than 5. The next step is to encode the transitive closure, and check its performance.

```
:- table reachable_full/2.
reachable_full(S,D) :- edge(S,D).
reachable_full(File1':'Source,File2':'Dest) :-
        reachable_full(File1':'Source,_':'Temp),
        reachable_full(_':'Temp,File2':'Dest).
```

With relevant calls to `findall/3` and `qtime/2` we can find out the total number of `reachable_full/2` edges the transitive closure of the graph contains, and how much time is needed to run through the graph:

```
|?- findall(_,reachable_full(_,_),L),length(L,N).
N = 571295
```

```
|?- qtime(reachable_full(S1':'F1,S2':'F2),T).
T = 10.4210
```

The total size of the transitive closure of the initial graph is roughly 570,000 edges, which is 10 times the number of the original graph, and it takes about 10 seconds to go through the transitive closure of the original graph.

An interesting characteristic of this implementation is that as users issue more queries, future queries will be answered more quickly. This is because additional queries will populate the `reachable_full/2` table, which effectively memoizes the results for future queries and can be checked in constant time. For example, `kmalloc()` is a widely-used function within the Linux Kernel, so presenting some information about it gives us a rough estimate of how long will it take for large enough queries to be completed:

```
|?- findall(_,reachable_full(_':'_,_':'kmalloc),L),length(L,N).
N = 8032
```

```
|?- qtime(T,reachable_full(_':'_,_':'kmalloc)).
T = 4.8000
```

```
|?- qtime(T,reachable_full(_':'_,_':'kmalloc)).
T = 0.0000
```

The first query of all the functions that call `kmalloc()` takes almost 5 seconds; yet the second query is effectively instant (constant time). What is remarkable about this behavior is that we get it in XSB-Prolog by just using the `:- table reachable_full/2` directive. Finally, had we not indexed the `edge/2` facts, a call to go through the transitive closure of the original graph would have been again a factor of 5 slower, as the following query reveals:

```
| ?- qtime(T,reachable_full(_,_)).
T = 65.1190
```

6   An Interface for Users

The last component of this framework is an interface between the engine, described in the previous 2 sections, and users. Rather than requiring users to install XSB-Prolog and issue Prolog queries directly, we created a PHP website for users to issue queries to the database and to display the answers. Our tool provides 4 different pre-compiled queries to the transitive closure of the `edge/2` relation:

1. Provided a *filename*, retrieve function call dependencies within the filename
2. Provided a *source* function name, retrieve the names of all the functions that are called (directly and indirectly) from it

3. Provided a *destination* function name, retrieve the names of all the functions that call it (directly and indirectly)
4. Provided a function name and a (possibly empty) list of other functions, retrieve the names of all functions that the former calls which are not contained in the later (this is also called a *cut-off*)

6.1   Pre-compiled Queries

Since we decided that the results of these queries will be provided to users via a web-browser, the output should be a specific HTML string that represents the information in an understandable manner. The `write_html/4` predicate takes as arguments the name of the operation (`file`, `source`, `dest` or `cut_off` as described in the list above), the respective filename or source/destination/cut-off function names, and in the cut-off case a (possibly empty) list of functions we have already implemented. The fourth argument will be eventually bound to specific HTML string that corresponds to the information we wish each query to provide.

6.2   The Server, Client and Web Interface

At this point, we have all of the infrastructure needed to retrieve function call-dependency information from the Linux Kernel, process it and present the results in a user-friendly manner. The only pieces needed are a server that will receive query requests from clients, a `PHP` website that will be the interface between the users and the engine, and a client program that will communicate the necessary requests to the server, receive the answers and present them to the user. Both the server and client are written in XSB-Prolog, and the communication between them is implemented using the `socket.P` library.

This server code is included in the same source file as all the code we presented above; it is necessary that the `edge/2` relation and its transitive closure are kept in the server's memory at-all-times if we want to take advantage of the tabling capabilities of XSB. Calling the `server/0` predicate will initialize the server and keep it running forever listening for requests from clients. Once a request is received from a socket, the respective goal is called, an answer list is constructed with an appropriate call to `findall/3` and is returned to the client via another socket.

We call the `client/1` predicate with an appropriate argument, which will be exactly the goal we want the server to call and give us answers for. This goal is constructed on-the-fly inside the PHP script located at http://ewl.cewit.stonybrook.edu/spyros/kernel.php, based on the users' selections. This script defines the various options users have for querying the server. According to which option the user chooses, and which arguments she provides, a query string is built on-the-fly, which calls the client code mentioned above. In turn, the client code communicates the request to the server, receives the answer when it is computed and presents the result to the user.

7   A Note on Efficiency

This section presents running times and memory consumption for specific queries to our engine. These statistics demonstrate the necessity of tabling and indexing in the backing Prolog engine, even for medium-sized data analysis. The **Query** column lists the query in question; the **Tabling** and **Indexing** columns list tabled and indexed predicates, respectively, that we are interested in (and No if none is tabled); the **Time** column lists the time needed to compute the query in seconds; and the **Memory** column lists the space allocated at the end of the computation in megabytes. We compute the memory consumption using XSB's `statistics/0` predicate, by subtracting the memory consumption when XSB is initialized (approximately 1MB) by the memory consumption in the end of every computation. A '-' entry in the table means that for the particular query that option is not meaningful. The "default" entry in the **Indexing** column means that trie indexing was not enabled for the `edge/2` facts, hence first-argument indexing was used, which is the default indexing mechanism XSB uses for all terms.

Table 1 shows information regarding parsing and walking the graph and its transitive closure. Tables 2 and 3 show information regarding running the query `q1` (see Table 4). In Table 2, only a call to `parse_files/0` was made before taking the measurements, whereas in Table 3, calls to `infor` and `infoe` queries (see Table 4) were made before taking the measurements.

Query	Tabling	Indexing	Time (s)	Memory (Mb)
`parse_files/0`	-	default	111.8580	26.89
`parse_files/0`	-	`edge/2`,trie	19.3700	22.32
`infoe`	-	default	0.0170	22.32
`infoe`	-	`edge/2`, trie	0.0170	22.32
`infor`	No	default	71.6620	131.02
`infor`	No	`edge/2`, trie	14.5410	127.11
`infor`	`reachable_full/2`	default	67.2490	131.17
`infor`	`reachable_full/2`	`edge/2`, trie	14.2860	127.07
`infor`	`reachable_full/2`	`edge/2`, trie `reachable_full/2`, 1+2	14.3530	127.71

Table 1. Parsing and Walking through the graph

The results presented in the tables above show some interesting facts regarding the suitability of Prolog systems to handle and process medium-sized datasets. Despite the fact that the Prolog code we wrote for developing this tool is simple, compact and easy to maintain, had XSB not offered tabling and indexing mechanisms, it would be unsuitable for processing even medium-sized graphs. Indexing speeds-up the time needed to parse and assert the facts by a factor of at least 5 in each case, and query processing time by a factor of almost

Query	Tabling	Indexing	Time (s)	Memory (Mb)
q1	No	default	131.0560	97.11
q1	reachable_full/2	default	121.0490	97.23
q1	No	edge/2, trie	15.8320	92.97
q1	reachable_full/2	edge/2, trie	15.6110	92.55
q1	reachable_full/2	edge/2, trie reachable_full/2, 1+2	15.2670	92.52

Table 2. Query time/memory information without calls to `infor` and `infon`

Query	Tabling	Indexing	Time (s)	Memory (Mb)
q1	No	default	57.0660	135.75
q1	reachable_full/2	default	64.1570	135.65
q1	No	edge/2, trie	4.8150	131.84
q1	reachable_full/2	edge/2, trie	5.7140	131.25
q1	reachable_full/2	edge/2, trie reachable_full/2, 1+2	4.9440	131.80

Table 3. Query time/memory information after calls to `infor` and `infon`

```
infoe findall(_,edge(_,_),List), length(List,N)
infor findall(_,reachable_full(_,_),List), length(List,N)
q1    findall(_,write_html(dest,kmalloc,_,Ans),L)
```
Table 4. Legend

8. Tabling doesn't make any noteworthy changes to the speed of answering a query for the first time, but it reduces the effort of retrieving an answer already computed to a *constant* time. Moreover, if we pre-run some queries like `infoe` and `infor` before launching the server, building queries for the first time is even faster.

Finally, memory consumption is not an issue for this tool. Commodity systems often have several GB of RAM, and our tool used at most 140MB when processing a graph of more than half a million edges.

8   Conclusion and Future Work

In this paper, we investigated modern Prolog systems for building tools able to handle data representing medium-sized graphs. Our first such tool is used to retrieve function dependencies from the Linux Kernel to help developers understand complex attributes of the system. The methods we described in the previous sections are not only applicable to the Linux kernel; as a matter of fact, *any* C code base can be the subject of the function dependency analysis we described. Moreover, we have shown that tabling and indexing play an *integral* role in such efforts, thus making XSB-Prolog suitable for applications that were gen-

erally out of Prolog's "sweet spot". All the code described in this paper, along with the appropriate dataset obtained by compiling the Linux Kernel can be found in http://www.cs.stonybrook.edu/~shadjichrist/kerfdep.tar.bz2.

There are many directions for future work based on this tool. Among the most challenging, would be figuring out the appropriate queries that can help automating the process of modularizing chunks of kernel code, instead of relying on a human expert. A quite straightforward query we can write in our current setting, would be one that includes in the *cut-off* functions that are being called from a particular one, but don't call other functions in their bodies. In this query, we are making the assumption that functions that do not have function calls in their bodies are implementing core functionalities in the Linux Kernel, and are particularly written this way for efficiency reasons. Although this can be a good starting point for automating the process of modularizing code in the kernel, more complex analyses should be used to get as close to what the human expert would decide as possible.

One other possible path would be the integration with popular source code editors (such as Emacs, Vim, Eclipse), which would enable kernel developers to use it more easily. With the use of *Tabling with Answer Subsumption* [23] our framework is able to easily answer quantitative queries regarding function dependencies, like for example "which is the most heavily used function in the Linux Kernel", or "which is the function mostly called from function A". Having this kind of statistics in hand, will enable kernel developers to focus their optimizations to particular heavily used functions and components. Finally, larger datasets which contain much richer information regarding the kernel's behavior (like for example LXR's dataset) can be used.

9   Acknowledgments

References

1. Colmerauer, A., Roussel, P.: The birth of Prolog. In: History of programming languages—II, ACM (1996) 331–367
2. Covington, M.A.: Natural Language Processing for Prolog programmers. Prentice Hall Englewood Cliffs (NJ) (1994)
3. Rao, P., Sagonas, K., Swift, T., Warren, D.S., Freire, J.: XSB: A system for efficiently computing well-founded semantics. In: Logic Programming And Non-monotonic Reasoning. Springer (1997) 430–440
4. Saha, D.: Incremental evaluation of tabled logic programs. PhD thesis, Stony Brook, NY, USA (2006) AAI3258884.

5. Saha, D., Ramakrishnan, C.: Symbolic Support Graph: A space efficient data structure for incremental tabled evaluation. In: Logic Programming. Springer (2005) 235–249
6. Merritt, D.: Building expert systems in Prolog. Springer-Verlag New York, USA (1989)
7. Laera, L., Tamma, V., Bench-Capon, T., Semeraro, G.: SweetProlog: A system to integrate ontologies and rules. In: Rules and Rule Markup Languages for the Semantic Web. Springer (2004) 188–193
8. Papadakis, N., Stravoskoufos, K., Baratis, E., Petrakis, E.G., Plexousakis, D.: PROTON: A Prolog Reasoner for Temporal ONtologies in OWL. Expert Systems with Applications **38**(12) (2011) 14660–14667
9. Stickel, M.E.: A Prolog technology theorem prover: Implementation by an extended Prolog compiler. Journal of Automated reasoning **4**(4) (1988) 353–380
10. Lally, A., Prager, J.M., McCord, M.C., Boguraev, B., Patwardhan, S., Fan, J., Fodor, P., Chu-Carroll, J.: Question analysis: How Watson reads a clue. IBM Journal of Research and Development **56**(3) (2012) 2
11. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM (2011) 1213–1216
12. DLV: http://www.dlvsystem.com/company
13. SEMMLE: http://www.semmle.com/
14. Ramakrishnan, C., Ramakrishnan, I., Warren, D.S.: Xcellog: A deductive spreadsheet system. Knowledge Engineering Review **22**(3) (2007) 269–280
15. Becker, M.Y., Fournet, C., Gordon, A.D.: Secpal: Design and semantics of a decentralized authorization language. In: Proc. IEEE Computer Security Foundations Symposium. (2006)
16. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: Proceedings of the ACM SIGMOD International Conference on the Management of Data, Citeseer (1994) 442–453
17. Tweedie, S.: Ext3, journaling filesystem. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
18. Olson, J.: Enhance Your Apps With File System Transactions. MSDN Magazine (July 2007) http://msdn2.microsoft.com/en-us/magazine/cc163388.aspx.
19. Spillane, R., Gaikwad, S., Chinni, M., Zadok, E., Wright, C.P.: Enabling Transactional File Access via Lightweight Kernel Extensions. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST). (2009) 29–42
20. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SOSP (2003)
21. McPherson, A.: A conversation with Chris Mason on btrfs: the next generation file system for Linux. http://www.linuxfoundation.org/news-media/blogs/browse/2009/06/conversation-chris-mason-btrfs-next-generation-file-system-linux
22. McKenney, P.E.: Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels. PhD thesis, Oregon Health and Science University (2004)
23. Swift, T., Warren, D.S.: Tabling with answer subsumption: implementation, applications and performance. In: Logics in Artificial Intelligence. Springer (2010) 300–312