Development and Architecture of REFPERSYS
A Multi-Threaded REFlective PERsistent SYStem for AI

Basile Starynkevitch[1] and **Abhishek Chakravarti** [2] and **Nimesh Neema**[3]
`refpersys.org`

Abstract.

The technical progress of computing hardware, especially with the prevalence of multicore systems and large amounts of RAM, now allows us to further experiment with the Artificial General Intelligence goals inspired by the works of pioneers such as J. Pitrat[10, 9, 11]. Our project is exploring the development, through the use of metaprogramming approaches, a bootstrapped multithreaded, reflexive and orthogonally persistent Quine system running on modern Linux x86-64 hardware, leading to a declarative knowledge-based language.

Topics : Knowledge Representation and Reasoning, Machine Learning, Multidisciplinary Topics and Applications, Agent-based and Multi-agent Systems, Semantic Technologies.

1 MOTIVATIONS

A symbolic AI software system running on LINUX nowadays needs to manage a large amount of information and knowledge organized as some complex graph (also known as an *ontology* [8], a *semantic network* [17], or a *frame* [1, 7]) inside the computer memory. This large graph should be orthogonally persistent [4] (such as in BISMON [15]) on disk, and be loaded from files at startup time on mornings of worked days, and later dumped to disk, as a set of *state files* in HJSON format, before normal termination when leaving office at evening. Having these files in a textual format facilitates their management with existing distributed version control systems such as `git` or software development forges like `gitlab` and ensures some data portability. Carefully generating the runtime, through metaprogramming approaches, as *C++* code (à la GCC MELT [14] that is subsequently compiled by `g++` into `dlopen`-ed plugins[4] provides some amount of code portability. Current desktop computers are powerful enough to keep a large memory heap in RAM[5], and this entire heap can be persisted on disk, similar to how database management systems such as SQLITE or *NoSQL* work.

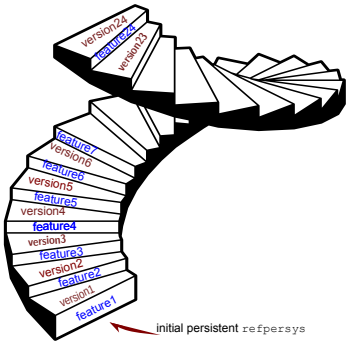
Current processors are multi-core, so running a good enough multi-threaded program on them [3] should be beneficial for performance. Backtracing libraries such as I. Taylor's `libbacktrace` facilitate introspection of the current thread's call stack, if the *C++* code has been compiled, using `g++ -O2 -g`, with DWARF debugging information. JIT-compiling libraries (such as `libgccjit`) can produce plugins without the need of a textual representation of some AST of the code.

[1] Bourg La Reine, France, email: `basile@starynkevitch.net`
[2] Kolkata, India, email: `chakravarti.avishek@gmail.com`
[3] Indore, India, email: `nimeshneema@gmail.com`
[4] In practice, as demonstrated by the `manydl.c` sample code written for J. Pitrat, on `github.com/bstarynk/misc-basile/`, several hundreds of thousands of `*.so` plugins can be generated and then dynamically loaded by `dlopen` at runtime on LINUX desktops.
[5] Notice that the millions of SLOC for mature software (such as the GCC compiler, the QT GUI toolkit, or the POSTGRESQL database), fits entirely in the 64 GB RAM of a powerful desktop. But compiling such a code base takes hours of computer time.

2 THE REFPERSYS ROADMAP

2.1 A Staircase Development Model

REFPERSYS development model is similar to a staircase, as depicted in Figure 1, unlike the traditional spiral development model [2]. The floor of the staircase is just a C++ hand-coded persistent system, and we gradually add new code implementing more features (first entirely hand-written, later more and more parts of it being replaced by REFPERSYS generated code). We are progressively replacing existing hand-written code (or low-level DSL) by more a expressive and generated one. So we will continuously rewrite past formalizations as more clever and expressive ones, taking increasing advantage of REFPERSYS system-wide introspective and metaprogramming abilities.



Each new feature -or small incremental change or a few of them (small `git commits`) - of REFPERSYS enables us to build and **generate** the next version of REFPERSYS, and a next feature is then added to that *improved* version, and so on repeatedly, etc....

Figure 1. the strange **REFPERSYS staircase development model** (from a figure of Spiral stairs by Lluisa Iborra from the Noun Project)

So the REFPERSYS project is taking a bootstrapping approach [9, 10, 5] : progressively old code (perhaps even hand-written, or generated) is replaced by "better" code emitted by metaprogramming techniques from higher level formalizations.

2.2 Initial Architecture of REFPERSYS

The initial architecture[6], prototyped in C++17, of REFPERSYS is close to BISMON's one. But it should evolve very differently. Our persistent and garbage-collected [6] heap is made of *values*. Most values are immutable and rather light. Some values are mutable *objects*, which are quite heavy, since synchronized between threads so carrying their read-write lock. Values are represented in 64 bits machine words: either a tagged integer, or containing a pointer to some aligned memory zone. Most values are persistent—so dumped then later reloaded through state files—but some are transient, since it makes no sense to keep them on disk. Transient values, often transient objects, include reification of GUI windows or Web widgets, HTTP connections, ongoing processes, in particular compilation commands of newly generated plugins, etc.. Values are both ordered and hashable, so fit nicely inside standard C++ containers like `std::set` or `std::unordered_map`. Every mutable object has a globally unique, fixed, and random *objid*, which fits in 16 bytes and is textually represented—in state files—with a string such as `_7VnQtHZ63pA02rCekc`.

Immutable values include UTF-8 strings, boxed IEEE 64 bits floats without `NaN` to stay ordered, tuples of references to objects, ordered sets of objects, closures -whose code is represented by some object, and with arbitrary values as closed values-, and immutable instances.

Mutable objects carry their constant *objid*, their lock, their class -which could change at runtime and is an object-, attributes, components, and some optional smart `std::unique_ptr` pointer to the payload of that object. An attribute associates a key -itself some object reference- to a value, so attributes are collected in some mutable C++ `std::map`. The components are organized as a `std::vector` of values. The payload belongs to its owning object and carry extra data, such as mutable hashed sets, class information -sequence of superclasses and method dispatch table-, string buffers, opened file or socket handles, GUI or widgets etc..

REFPERSYS will initially have an ad-hoc IDE—built with the FLTK toolkit—to just fill the persistent heap and generate some of its C++ code. This IDE will support the syntax highlighting, autocompletion and navigating of objects through their *objids*.

3 METAPROGRAMMING IN REFPERSYS

An essential insight of REFPERSYS is metaprogramming, practically done by generating *C++17* code at runtime for a Linux system. This is strongly inspired by previous work, see [9, 10, 15, 14, 13]. The choice of the actual programming language used to generate code[7] in within REFPERSYS is mostly arbitrary and guided by non-technical concerns: which programming language is known to all the REFPERSYS team, while being compatible with a lot of existing open source libraries and APIs? That programming language happens to be C++ (better than C, because of its containers; also used in TENSORFLOW or GHUDI), but our expansion machinery is inspired by MELT code chunks [14], LISP macros [12] or DJANGO templates, driven by "expert system"-like meta rules (such as in [9]) potentially applicable to themselves.

[6] The GPLv3+ code of BISMON, mostly in C, is available on `github.com/bstarynk/bismon/`. But REFPERSYS is coded in C++, only for LINUX/X86-64, on `gitlab.com/bstarynk/refpersys` and share almost no code with BISMON.

[7] In practice, some C++ code is emitted in a file similar to `/tmp/generated.cc`, compiled as a plugin by forking `g++ -O -g -fPIC -shared` into a `/tmp/generated.so`, which is later `dlopen`-ed, all by the same process running the `./refpersys` executable.

4 CONCLUSION

We have discussed how we are trying to develop REFPERSYS organically, using metaprogramming techniques to eventually build a fully bootstrapped Quine system. Our approach is to gradually replace hand-written code with increasingly expressive generated code, relying on the growing metaprogramming and reflective properties of the system. See also [16].

ACKNOWLEDGEMENTS

Thanks to François Bancilhon for proof-reading this draft.

REFERENCES

[1] Daniel G. Bobrow and Terry Winograd, 'An overview of KRL, a knowledge representation language', *Cognitive Science*, **1**(1), 3–46, (1977).

[2] Barry W Boehm, 'A spiral model of software development and enhancement', *Computer*, 61–72, (1988).

[3] David R Butenhof, *Programming with POSIX threads*, Addison-Wesley Professional, 1997.

[4] Alan Dearle, Graham N. C. Kirby, and Ronald Morrison, 'Orthogonal persistence revisited', *CoRR*, **abs/1006.3448**, (2010).

[5] Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Stéphane Ducasse, Noury Bouraqadi, and Pablo Tesone, 'Challenges in Debugging Bootstraps of Reflective Kernels', in *IWST19 - International workshop on Smalltalk Technologies*, Cologne, Germany, (August 2019).

[6] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: the art of automatic memory management*, Chapman and Hall/CRC, 2016.

[7] Douglas B Lenat, 'Theory formation by heuristic search: The nature of heuristics ii: background and examples', *Artificial Intelligence*, **21**(1-2), 31–59, (1983).

[8] Antonio De Nicola, Michele Missikoff, and Roberto Navigli, 'A software engineering approach to ontology building', *Information Systems*, **34**(2), 258 – 275, (2009).

[9] Jacques Pitrat, 'Implementation of a reflective system', *Future Generation Computer Systems*, **12**(2), 235 – 242, (1996).

[10] Jacques Pitrat, *Artificial Beings: The Conscience of a Conscious Machine*, Wiley ISTE, 2009.

[11] Jacques Pitrat, 'A step toward an artificial artificial intelligence scientist', Technical report, CNRS and LIP6 Université Paris, (2009).

[12] Christian Queinnec, *Lisp in Small Pieces*, Cambridge University Press, New York, NY, USA, 1996.

[13] Basile Starynkevitch, 'Multi-stage construction of a global static analyzer', in *GCC summit*, pp. 143–152, Ottawa, Canada, (july 2007). GCC.

[14] Basile Starynkevitch, 'MELT - a translated domain specific language embedded in the GCC compiler', in *DSL2011 IFIP conf.*, Bordeaux (France), (September 2011).

[15] Basile Starynkevitch, 'Specialized static analysis tools for more secure and safer IoT software development. bismon documentation', Technical report, CEA/LIST, (oct. 2019). work in progress, H2020 draft deliverable.

[16] Basile Starynkevitch, Abhishek Chakravarti, and Nimesh Neema, 'REFPERSYS high-level goals and design ideas', Technical report, `refpersys.org`, (2019).

[17] R.P. Van De Riet, 'Linguistic instruments in knowledge engineering', in *Proc.1991 Workshop on Linguistic Instruments in Knowledge Engineering*. North Holland, (1992).

Our draft `git` ID is *`fdb271193086e90b...`*