Design ideas and implementation issues in REFPERSYS, an open source reflexive persistent system

Basile Starynkevitch, Abhishek Chakravarti

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. 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.**

Index Terms—**Agent-based and Multi-agent Systems, Knowledge Representation and Reasoning, Machine Learning, Multidisciplinary Topics and Applications, Semantic Technologies.**

I. INTRODUCTION

Our complex, but fragile, world is facing dramatic and extremely challenging planet-wide issues, such global warming, demographic and political crises, economic and financial emergencies, and growing inqualities. In the light of such challenges, **A**rtificial **G**eneral **I**ntelligence (*AGI*) systems are increasingly relevant.

We believe in free software (read also this), and we strongly believe that an AGI prototype should be some free software, exactly like most infrastructure software are (notably LINUX). See also the SOFTWARE HERITAGE project for interesting insights. REFPERSYS wants to be an AGI infrastructure , and there is work for many years (several years of work needed without any "artificial intelligence", just for the infrastructure).

An even partially successful AGI system might be useful to coordinate, run and manage other existing software (described through some knowledge given declaratively). Imagine how complex future digital twins of the entire planet Earth, designed to tackle with global warming, would need to be. For such dramatically complex usage, an AGI system (like REFPERSYS, if we succeed in making it) could be quite helpful to just drive and use such a "digital twin" simulation. Making it free software runnable on a free software operating system should benefit most of humanity (but keeping it proprietary won't), and enable further or alternative experimentations. And "there is no planet B". So investing a few persons willing to working for nearly a decade is not too much for such a perspective.

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* **DeNicola:2009:OntologyBuilding**, a *semantic network* **VanDeRiet:1992:Ling-instr-know**, or a *frame* **Bobrow-Winograd:1977:KRL**, **Lenat:1983:theory**) inside the computer memory.

This large graph should be orthogonally persistent **Dearle:2010:orthopersist** (such as in BISMON **Starynkevitch:2019:bismon-draft**) 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 JSON 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.

Current desktop computers are powerful enough to keep a large memory heap in RAM[1], 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 **butenhof:1997:programming** 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.

A. A Staircase Development Model

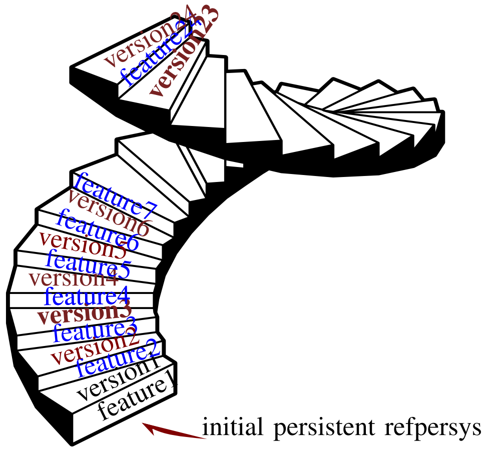
REFPERSYS development model is similar to a staircase, as depicted in Figure 1, unlike the traditional spiral development model **boehm:1988:spiral**. The floor of the staircase is just a C++ hand-coded persistent system, and we gradually add new code implementing more features (first entirely hand-written, 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[2]. So we will continuously rewrite

Contacts: `basile@starynkevitch.net` (Bourg-la-Reine, near Paris, France) and `abhishek@taranjali.org` (Kolkata, India), and collectively `team@refpersys.org`

[1]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]Currently, as of August 2020, REFPERSYShas 183 lines of generated code as compared to 17,337 lines of hand-written code.

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....

Fig. 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[3] approach **Pitrat:1996:FGCS**, **Pitrat:2009:ArtifBeings**, **hernandez-phillips:2019:debugging-bootstrap** : progressively old code (perhaps even hand-written, or generated) is replaced by "better" code emitted by metaprogramming techniques from higher level formalizations.

B. *Initial Architecture of* REFPERSYS

The initial architecture[4], prototyped in C++17, of REFPERSYS is close to BISMON's one. But it should evolve very differently. Our persistent and garbage-collected **jones:2016:gchandbook** 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*.

C. *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 **Pitrat:1996:FGCS**, **Pitrat:2009:ArtifBeings**, **Starynkevitch:2019:bismon-draft**, **Starynkevitch-DSL2011**, **Starynkevitch:2007:Multistage**. The choice of the actual programming language used to generate code[5] 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 **Starynkevitch-DSL2011**, LISP macros **Queinnec:1996:LSP** or DJANGO templates, driven by "expert system"-like meta rules (such as in **Pitrat:1996:FGCS**) potentially applicable to themselves.

II. DESIGN AND DEVELOPMENT

A. *Core Ideas*

REFPERSYS is a long term risky research project with an open science mindset and reproducible experiment ethics zuboff:2015:big-other, oneil:2016:weapons, and a

[3]By bootstrapping, we mean in the sense of a self-compiling compiler, and not as in the statistical sense of the term.

[4]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.

[5]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.

free software licensed under GPLv3+, and targetted *only* for LINUX X86-64 computers.. A Linux system with at least 16 Gibytes of RAM, 4 *x86-64* cores, and 220 Gibytes of disk is required. The grand ambition of REFPERSYS is to become later an infrastructure for some strong AGI system à la CAIA by Jacques Pitrat **Pitrat:1996:FGCS**, **Pitrat:2009:AST**, **Pitrat:2009:ArtifBeings**, but before even approaching that goal a lot of work is required, and REFPERSYS should be valuable by itself for other less ambitious and more pragmatic purposes, perhaps some specialized collaborative web server (GPLv3+) to ease communication between human REFPERSYS developers, that is a mix of a wiki, a chat, and a tool for sharing document with drawings or graphics.

The development of REFPERSYS is (like the one of `bismon`, or of CAIA) a slow, incremental and gradual bootstrapping process with a meta-programming **dormoy:1992:meta**, **hernandez-phillips:2019:debugging-bootstrap** approach : features added to REFPERSYS in January 2020 are used to implement new features worked on a later REFPERSYS in March 2020.

As every practical software, REFPERSYS targets some defined machines: common Linux distribution running on some computer[6]. So the target machine of REFPERSYS is a quite complete and modern Linux system (such as a recent DEBIAN or UBUNTU desktop), with many useful packages, and administered by some human person. The REFPERSYS system is published in "source" form, as a set of `git` versioned[7] textual files (e.g. hopefully generated *C* files[8], perhaps some `Makefile` or better yet an OMAKE build -most and more and more of them being generated- or shell files or data files). Some of these files are generated, and the bootstrapping goal is to have *every* `git`-registered textual file been generated by REFPERSYS, with a **bootstrap**ed approach[9] similar to those of self-hosting compilers.

Within REFPERSYS, we call "source file" any Linux file which is `git`-versioned. We hope that more and more of these source files will be generated by the `refpersys` ELF executable program. **A significant milestone is the entire bootstrapping of REFPERSYS**, when all files (in textual form, to stay `git`-friendly, like text based protocols are more friendly for developers) can be regenerated by the `refpersys` executable, exactly in the same state as they were previously[10] : as a whole, our REFPERSYS system should become a Quine program, and CAIA is already one. So the build automation tool which compiles REFPERSYS should use file contents, not modification times to trigger compilation commands, since a full regeneration of such a bootstrapped REFPERSYS system will touch all files, without changing the content of any of them. Hence and very concretely, for building REFPERSYS the `OMake` build automation tool is preferable to GNU `make`.

For pragmatical reasons, **REFPERSYS needs a good garbage collector** (or GC **appel:1991:garbage**, **wilson:1992:uniprocessorgc**, **baker:1995:cons**, **jones:2016:gchandbook**), since fully compile-time GC **mazur:2004:compile** are too difficult to implement. Since multi-core x86-64 machines are very common, it should take advantage of them, so **REFPERSYS should follow a multi-threaded approach** above POSIX **barney:2010:pthreads** or C++11 threads. Our GC should be a precise garbage collector **Rafkind:2009:PreciseGC** and we may want to favor, like what was done in GCC MELT **Starynkevitch:2007:Multistage**, **Starynkevitch-DSL2011**, **Starynkevitch-GCCMELTweb**, fast allocation of small memory zones which get quickly disposed of when becoming dead using a copying generational Cheney-like GC algorithm **wilson:1992:uniprocessorgc**. But mixing precise, sometimes generational GC techniques with multi-threading is a difficult programming task. But precise-GC friendly programming is simpler in generated C or C++ code that with hand-written code (because of explicit management of local GC roots and write barriers, à la QISH or OCAML: garbage collection invariants are boring and brittle to maintain in hand-written code).

Reification is an important concept in REFPERSYS, including (later) at the knowledge representation level with semantic networks and frames. REFPERSYS call stacks are made of call frames known to our garbage collector (like OCAML's ones). They could later be copied into data structures representing some delimited continuations **Reynolds:1993:continuations**, **Queinnec:2004:ContinWeb**, perhaps even representing and describing control **fouet-starynkevitch:describing-control:1987**, **Starynkevitch-1990-EUM**, **Pitrat:2009:ArtifBeings**. This should also enable **introspection**, by permitting primitives inspecting the current call stack, perhaps using Ian Taylor's `libbacktrace`. Also, such an introspection might perhaps be implemented **mitchell:2001:alp** with two nearly twin `refpersys` processes, one of them driving a `gdb` process[11].

REFPERSYS should (like CAIA and its predecessor MALICE did **Pitrat:2009:AST**, **Pitrat:1996:FGCS**, **Pitrat:2009:ArtifBeings**) have some expert system shell **kumar:2015:importance-expert-systems**, **nigro:2008:meta** and meta-rules to "dynamically compile" some subset of expert system rules and knowledge bases to procedural code (e.g. with a metaprogramming approach of generating *C* code, or `libgccjit` compiled code, then `dlopen(3)`-ing that code and running it at runtime. The `manydl.c` program show

[6]For several years, that computer is a desktop or powerful laptop running some DEBIAN. Later that could be some "virtual machine" e.g. some DOCKER container.

[7]We crucially depend upon `git` *specifically* (e.g. `GitLab`), and porting REFPERSYS to some other versioning system -or to some other operating system than LINUX- would be a quite difficult task.

[8]However, notice that bootstrapped language implementations like Scheme 48 or OCaml are keeping some bytecode form under version control, and CHICKEN SCHEME is, like `bismon`, `git`-keeping generated C files.

[9]Observe that Linux source distributions like `linuxfromscratch.org`, or to a lesser extent GenToo, are also, when considered as a single system, fully bootstrapped.

[10]Pedantically, some fixpoint of some very coarse-grained operational semantics related to abstract interpretation and big step semantics, each big step being the entire regeneration of the system, inspired by Futurama projections and partial evaluation.

[11]Imagine some `popen` or some `g_spawn_async` or some `Poco::Process` of some `gdb refpersys 1234` process debugging the other one of pid 1234.

that this can practically be done many dozen of thousands of times on Linux desktops).

REFPERSYS will extensively use **metaprogramming** techniques, so it **should generate code** (like CAIA do) in a transpiler approach (in C, C++, -compiled into plugins and later dynamically loaded with `dlopen(3)` - maybe also JavaScript and HTML5 if we decide to have a web user interface). REFPERSYS could also later use just-in-time compilation libraries such as `libgccjit`. The domain-specific language of REFPERSYS[12] (a declarative one, with "expert system rules") should gradually increase its expressiveness and become more and more declarative and closer to mathematical formalisms.

Most Linux distributions contain lots of useful libraries or software components for REFPERSYS long-term goals, notably machine learning open source libraries like TENSORFLOW **charniak:2019:deep-learning** or GUDHI **chazal:2016:high**. We might at some point also need messaging libraries like 0MQ, graphical user interfaces libraries à la QT or more probably web servicing libraries like `libonion` or WT. To decrease efforts, we don't want to rewrite such libraries inside REFPERSYS (considered as a very high level, declarative, domain-specific language). Hence, we will need in REFPERSYS to generate some glue code, like SWIG does, from some **declarative description** (probably some frames or knowledge bases) of the API of these available libraries.

REFPERSYS should at first be **orthogonally persistent**. Like BISMON **Starynkevitch:2019:bismon-draft** it will load its state (its entire garbage-collected heap) from files at startup, and will dump its state[13] into files at shutdown. These state files are textual, in JSON format, and `git`-versioned, and should be portable to other 64 bits Linux computers. A manifest file describing the collection of files keeping the state is probably needed.

B. Development Cycle

Ordinary software projects tend to follow a spiral development model **boehm:1988:spiral**. But REFPERSYS' development follows a strange loop **hofstadter:2007:strange-loop**, since it is bootstrapped in an evolutionary prototyping manner. It is more like a spiral staircase like in figure 1. The initial (floor) is just a persistent system, and we gradually add new code implementing more features (first entirely hand-written, later more and more parts of it replaced by REFPERSYS generated code). Of course the fun is in replacing existing hand-written code (or low-level DSL) by more expressive and generated one. So we will continuously rewrite past formalizations as a more clever and expressive ones, taking more and more advantage of REFPERSYS whole-system introspective abilities. All of EURISKO **Lenat:1983:Eurisko**, CYC **Lenat:1991:ev-cycl** and SELF[14] **chambers:1991:efficient** (or even IO or SMALLTALK) systems and their incremental development process are inspirational.

The first significant milestone of REFPERSYS should be the ability to re-generate all its textual source files (and maybe even `git add` then`git commit` them). That would require first implementing some simple template based machinery[15], withe the ability, like QUINE programs do, to regenerate all REFPERSYS source code (e.g. in C++, a `Makefile`, etc...). Actually REFPERSYS needs to conceptually have **self-modifying code Tschudin:2005:HarnessingSC**, practically implemented by systematically doing most function calls through indirect function pointers (which gets updated with `dlsym(3)`).

C. Metaprogramming and introspection approach

Metaprogramming is defined in Wikipedia as "a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running". That design idea is central to many Artificial Intelligence systems and AI inspired languages[16] and is also common in software engineering[17] **Lenat:1983:Eurisko**, **Lenat:1983:theory**, **Lenat:1991:ev-cycl**, **Pitrat:1996:FGCS**, **Pitrat:2009:AST**, **Pitrat:2009:ArtifBeings**, **Pitrat:blog**, **Queinnec:1996:LSP**, **Queinnec:2004:ContinWeb**, **Starynkevitch-1990-EUM**, **Starynkevitch-DSL2011**, **Starynkevitch-GCCMELTweb**, **Starynkevitch:2007:Multistage**, **Starynkevitch:2019:bismon-draft**, **Tschudin:2005:HarnessingSC**, **abelson:1996:sicp**, **briot:1987:uniform**, **chambers:1991:efficient**, **cointe:1987:metaclasses**, **dormoy:1992:meta**, **fouet-starynkevitch:describing-control:1987**, **greiner:1980:representation**, **hernandez-phillips:2019:debugging-bootstrap**, **hofstadter:2007:strange-loop**, **kay:1996:early-smalltalk**, **kelsey:1998:r5rs**, **kumar:2015:importance-expert-systems**, **matthews:2005:operational**, **mazur:2004:compile**, **nigro:2008:meta**, **queinnec:2003:lisp**, **Starynkevitch:2009:grow**, **serrano:1995:bigloo**. Generating some "source" code at build time is usual practice, advocated also by the NINJA build system, and theorized (around 1930, before even computers existed) in the CHURCH-TURING thesis. Related concepts include the famous (but undecidable) **halting problem** (whose proof involves a metaprogramming approach **Hofstadter:1979:GEB**), hygienic macros, and Rice's theorem.

Practically speaking **abelson:1996:sicp**, metaprogramming is easier achieved by explicitly representing (maybe incomplete) code with abstract syntax trees (or AST), maybe with

[12]That domain-specific language has to be defined and implemented in a bootstrapped manner.

[13]In a manner inspired by SBCL `save-lisp-and-die` primitive, or POLYML `export` primitive, or marshalling facilities of OCAML or PYTHON `pickle` module.

[14]SELF was even able (in hours of CPU time) to redefines its integers -even for arithmetic used inside its compiler- as bignums.

[15]Perhaps inspired by simple designs like DJANGO tempates, but driven by frame-based REFPERSYS objects.

[16]See also SCHEME 48, SBCL, RUST, even C++ *templates*, CHICKEN SCHEME, METAOCAML, the ECLiPSe Constraint Programming System, RASCAL, NEMERLE, COCCINELLE, OCSIGEN, GNU PROLOG, CLIPS, GPP, SWIG, ANTLR, IBURG, Gnu BISON, etc ...

[17]A typical example is the GCC compiler, or AUTOCONF, and transpiler approaches

some holes for metavariables for their later explicit substitution, in the spirit of DJANGO templates or of COMMON LISP macros or SCHEME macros. A practical way to implement such a template machinery for generating C or C++ code is given by GCC MELT code chunks **Starynkevitch-DSL2011**, **Starynkevitch-GCCMELTweb**, **Starynkevitch:2009:grow**, **Starynkevitch:2007:Multistage**, where a piece of C (or C++) code with holes (or metavariables) is passed through a "macro-string". Later, such a macro-string or code chunk can be expanded by filling the holes, that is expanding the metavariables (e.g.`$msg`) appropriately. Such an expansion might be recursive, since some hole filling (or metavariable replacement) could in turn trigger expansions of other macro-strings. In practice, REFPERSYS will use similar code chunks and macro-expansion to generate its C (or C++) code, and some initial ad-hoc integrated development environment (or IDE) will have to be coded, handling passively some persistent store. The expansion will be done through some scripting language (or *domain specific language*, a.k.a. DSL) which has to be implemented inside our IDE.

Metaprogramming involves code generation (using source-to-source ahead-of-time and/or just-in-time[18] compilation techniques **Aho:2006:dragon-book**, and in REFPERSYS is useful for many tasks, such as generating the garbage collection support routines for scanning or forwarding, and the loading and dumping routines needed for persistence (in the spirit of RPCGEN, SWIG and other serialization frameworks).

In REFPERSYS, metaprogramming is often and practically achieved (like in **Starynkevitch-DSL2011**, **Starynkevitch:2019:bismon-draft**, **Pitrat:1996:FGCS**, **Pitrat:2009:ArtifBeings** and our `manydl.c` example program), by generating some C or C++ code in a temporary file[19] like `/tmp/rpsgen123.c`, compiling that file **drepper:2011:write-shared-lib** into a generated plugin `/tmp/rpsgen123.so` by running a process such as `gcc -fPIC -Wall -O -g -shared /tmp/rpsgen123.c -lsomething -o /tmp/rpsgen123.so` and waiting for its successful completion, then `dlopen(3)`-ing that newly generated `/tmp/rpsgen123.so`, in a manner compatible with our garbage collection and agenda invariants. We might later care about carefully `dlclose(3)`-ing that generated plugin, but in practice we accept some limited virtual memory plugin leak, and we could just dump appropriately our persistent state by mentioning in some generated Manifest file those plugins which should be saved (as generated C code) with the state.

Reflection is "the ability of a process to examine, introspect, and modify its own structure and behavior" and also, for self-reflection, the capacity " to exercise introspection and to attempt to learn more about their fundamental nature and essence". (Wikipedia). It is advocated (in **Pitrat:2009:ArtifBeings**) that a similar approach is (painfully) achievable in AI systems, and it would need both clever backtracking and backtracing techniques. Libraries such as Ian Taylor's `libbacktrace` (which wants most of the code to be compiled with DWARF debugging information[20]) are helpful.

Our precise garbage collector (see §**??** below and **rafkind:2009:precise-gc**, or QISH) wants local variables holding garbage collected pointers to be known to the GC. In practice, the REFPERSYS call frame is some explicit local `struct` named _ in generated C code[21]. Such explicited local frames can often be optimized by `GCC` or `g++` (invoked with `-O2`).

As suggested by Pitrat (see **Pitrat:1996:FGCS**, **Pitrat:2009:AST**, **Pitrat:2009:ArtifBeings**), call stack reflection and backtrace is the elementary brick of more sophisticated *introspection* techniques. At some point, our REFPERSYS system should inspect its call stack and may take decisions after that. A typical approach would be to run such introspection once in a while (e.g. every 0.1 second on the average[22], in the inference engine of some expert system or knowledge base component of REFPERSYS.

Since we aim to be able to re-generate most (and hopefully all) of REFPERSYS code (in C or in C++), having simple **coding conventions** does matter: every REFPERSYS-defined C or C++ identifier should start with `rps_` in lower, upper, or mixed case (e.g. also `RPS_` or `Rps_`). Every C or C++ function, even `static inline` ones appearing in header files, has its name starting with `rps_` and is *globally* unique to the entire `refpersys` program. The C (or C++) code should be automatically indented[23] using Gnu INDENT or ASTYLE. Every named `struct` (in C) should have its tag matching `rps_*st`. Every `typedef`-ed data type should have its name matching `rps_*t`. Every named `enum` should have its tag matching `rps_*en` and the various enumerated values like `RPS_*`. Even in cases the C (or the C++) language allows several name spaces, we don't use that facility. Hence we avoid coding the common `typedef struct rpsfoo_t rpsfoo_t;` but prefer instead (inspired by GTK) coding `typedef struct rps_`*`foo`*`_st rps_`*`foo`*`_t`. Of course, names of local variables (that is automatic variables with their lexical scope limited to some small C or C++ block) could be as short as a single letter such as `i`. In general, our C or C++ code is written with the hope of being easily able to regenerate it.

D. The persistent heap

When REFPERSYS is running in some multi-threaded LINUX process, the REFPERSYS persistent heap is (like Bismon's one **Starynkevitch:2019:bismon-draft**) semantically

[18]Several JIT compilation libraries exist, notably `libgccjit` provided inside recent GCC compilers.

[19]There are practical reasons to generate these temporary files outside of `/tmp/`, which gets cleaned at reboot.

[20]In practice we should compile our or other C or C++ code with both `-O2 -g` passed while invoking `GCC` or `g++`, and this is indeed possible and practically works well enough.

[21]Like Bismon does, see its `LOCAL_BM` macro. See also the `CAMLparam`*`i`* and `CAMLlocal`*`j`* C macros of OCAML, and the `Py_VISIT` and `Py_DECREF` and other macros of PYTHON, the *foreign function interface* of SBCL, etc ...

[22]Timing considerations are essential, practically speaking, in REFPERSYS. See `time(7)` man page.

[23]With the social convention that REFPERSYS contributors are running `omake indent` or `make indent` before every `git commit`!

like the memory heap of most dynamic programming languages (such as PYTHON, GUILE, GO, SBCL, etc ...). We strongly want to avoid any GIL, but multi-threaded precise efficient garbage collector implementations are quite difficult to code. However, notice that the persistence (dump as textual `git`-versioned disk files) of a heap uses algorithms similar to those of copying garbage collectors **wilson:1992:uniprocessorgc**, **jones:2016:gchandbook**.

Conceptually, REFPERSYS tracing precise garbage collector should traverse the graph of references to REFPERSYS values, starting from global or transient roots and local variables inside call frames of working threads. Each REFPERSYS **value** (immutable or object) is represented by a machine word (aligned, 64 bits) which usually contains a pointer, but sometimes some tagged integer. Immutable values are often "small" (typically, less than a few dozens of words of memory, sometimes a lot more) but objects are necessarily heavier since they contain some kind of lock. closures are immutable values, containing an object representing and giving their function code (as a C function pointer inside that object), and additional closed values. In practice our garbage collector processes not only values (either immutable values or objects), but also **quasi-values** : these are a single memory zone which is allocated using the garbage collector allocation protocol, traversed by the GC when something points to it, appears inside other values (in particular, as payload of objects), but by convention should not be passed as a genuine value. Some values (or objects) may be dead and should eventually be reclaimed by the garbage collector.

Values, either immutable values or changeable objects, in REFPERSYS can be either **persistent** (dumped in textual state files[24], then reloaded at restart of `refpersys` process) or **transient** (that is, not dumped and not appearing in state files).

The **persistence** machinery - the dump - is conceptually simple and could run in several threads: start from global roots and traverse the memory graph but ignore transient objects and transient roots and memoize previously seen persistent objects. Of course, objects should not be persisted twice, and are referred by the **object id** or **objid** in the state files produced by the dump. That *objid* is alphanumeric, randomly generated and so hopefully globally unique -like `_2om48kc3k5R02d3ktW` for example- in our current implementation; exactly like UUIDS should be. Notice the conceptual similarity between REFPERSYS dump algorithm and its tracing garbage collector: both are traversing the graph of references inside the heap.

The global roots are objects. Use the C++ functions `rps_each_root_object` to iterate on them, `rps_add_root_object` to add one, `rps_remove_root_object` to remove one, `rps_is_root_object` to test if an object is a global root, `rps_set_root_objects` to get the set of all of them, and`rps_nb_root_objects` to get their number. Of course, some global roots can be transient objects, but all of them are roots for the garbage collector.

[24]In the current implementation, REFPERSYS state files should appear under `persistore/` subdirectory, and the manifest file is `rps_manifest.json` at the top directory.

The initial loading machinery (recreating a suitable heap - and rebuilding a graph of references inspired by figure **??**, without any transient stuff) from its previous dumped state) is first creating empty all objects, then later filling each of them. However, for efficiency, we may want to load the heap in parallel, using several loader threads. This could be easy if, after having created all objects as empty, and loaded plugins (i.e. `dlopen`-ing many `*.so` files), REFPERSYS processes each state file in a potentially different loading thread.

III. FUTURE POTENTIAL APPLICATIONS

The REFPERSYS project needs real-life applications, which could take advantage of its flexible object model, persistence, reflexivity and metaprogramming abilities. Future use-cases that are considered (within the next five years, if users are found) could include:

- representing wisely the documention of REFPERSYS, and generating some parts of it (in PDF or HTML format), probably starting a few LaTeX-related Linux processes and generating some `*.tex` or `*.html` files.
- assistance and software tool to help building, managing and technically coordinating a cooperating research consortium, such as European H2020 or future HorizonEurope[25] projects. Since the typical research consortium is made of many partner organizations, involves dozens of persons with various interests and skills.
- during the current Covid health crisis (and ignoring important legal obstacles or ethical concerns related to privacy issues - both should be handled by humans), managing data and information about Covid handling in a city: the set of tested persons, their medical results (represented as REFPERSYS objects, the set of testing centers, the databases holding test results, etc...
- in a complex industrial corporation (e.g. some automobile maker), facilitate the connection between existing software subsystems (following the "digital twins" dream), include finite elements simulation code, factory planning, etc. REFPERSYS could then help to build a semi-coherent and semi-automatic model of what is actually going on in such factories. Industrial parts, simulation software, databases could be represented as REFPERSYS objects
- etc...

IV. 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 **starynkevitch:2019:refpersys-design**.

ACKNOWLEDGMENT

The authors would like to thank Nimesh Neema for testing and minor contributions to REFPERSYS...

[25]See `ec.europa.eu/info/horizon-europe-next-research-and-innova`

Basile Starynkevitch Lives in France

Abhishek Chakravarti Lives in India

Our draft `git` ID is *`6f6c8f8d1f2a6f21`*.... Photos and biographies will be added later. Thi draft has been LaTeX-ed on *Mon 31 Aug 2020 11:56:01 AM MEST*.