SoftwareX 16 (2021) 100817

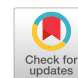
Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

2P-Kt: A logic-based ecosystem for symbolic AI



Giovanni Ciatto [a,*], Roberta Calegari [b], Andrea Omicini [a]

[a] Department of Computer Science and Engineering (DISI), Alma Mater Studiorum—Università di Bologna, Italy
[b] Alma Mater Research Institute for Human-Centered Artificial Intelligence (AlmaAI), Alma Mater Studiorum—Università di Bologna, Italy

ARTICLE INFO

Article history:
Received 11 January 2021
Received in revised form 8 August 2021
Accepted 13 September 2021

Keywords:
Logic programming
Artificial intelligence
Prolog
Kotlin
tuProlog

ABSTRACT

To date, logic-based technologies are either built on top or as extensions of the Prolog language, mostly working as *monolithic* solutions tailored upon specific inference procedures, unification mechanisms, or knowledge representation techniques. Instead, to maximise their impact, logic-based technologies should support and enable the general-purpose exploitation of all the manifold contributions from logic programming. Accordingly, we present 2P-Kt, a reboot of the tuProlog project offering a general, extensible, and interoperable *ecosystem* for logic programming and symbolic AI.

© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Code metadata

Current code version	`0.15.2`
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00007
Code Ocean compute capsule	–
Legal Code License	Apache License, Version 2.0
Code versioning system used	Git
Software code languages, tools, and services used	Kotlin Multiplatform, JVM, Node JS
Compilation requirements, operating environments & dependencies	Kotlin $\geq$ `1.4.0`, JDK $\geq$ `11`, Gradle $\geq$ `6.7`
If available Link to developer documentation/manual	http://pika-lab.gitlab.io/tuprolog/2p-in-kotlin
Support email for questions	giovanni.ciatto@unibo.it

Software metadata

Current software version	`0.15.2`
Permanent link to executables of this version	https://github.com/tuProlog/2p-kt/releases/tag/0.15.2
Legal Software License	Apache License, Version 2.0
Computing platforms/Operating Systems	Android, Linux, Mac OS X, Microsoft Windows, Unix-like
Installation requirements & dependencies	Java Runtime Environment (JRE) $\geq$ `11` or Node JS $\geq$ `12`, Kotlin v. `1.4.20`, ANTLR Runtime v. `4.8.*`, Java FX v. `15`, Kt-Math v. `0.2.6`, Clikt v. `2.8.0`
If available, link to user manual - if formally published include a reference to the publication in the reference list	http://pika-lab.gitlab.io/tuprolog/2p-in-kotlin
Support email for questions	giovanni.ciatto@unibo.it

The permanent link to the GitHub repository of 2P-Kt includes the `README.md` and `LICENSE` files. Source code is partitioned in many modules: each module has its own `src/` directory, within which source files are organised according to Gradle's convention. A tag corresponding with the version of the software to be reviewed – namely, `0.15.2` – is defined on the repository. Documentation generated from the `documentation/` module is publicly available on the Web.

1. Motivation and significance

Computational logic (CL) is a fundamental research area for *artificial intelligence* (AI), dealing with formal logic as a means for computing [1]. Its penetration into *symbolic AI* is nearly pervasive nowadays, and increasingly going deeper within *sub-symbolic*

* Corresponding author.
E-mail addresses: giovanni.ciatto@unibo.it (Giovanni Ciatto), roberta.calegari@unibo.it (Roberta Calegari), andrea.omicini@unibo.it (Andrea Omicini).

https://doi.org/10.1016/j.softx.2021.100817
2352-7110/© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

AI [2,3]: CL has enabled the development of the former in the past, and it is now pushing the latter towards interpretability and explainability. Be it either exploited to manipulate symbols or to make sub-symbolic solutions human-intelligible, the common expectation is that CL can endow software systems with *automated* reasoning.

Generally speaking, automated reasoning involves three major aspects: *(i)* logic, *(ii)* inference rule, and *(iii)* resolution strategy. *Logics* formally define how knowledge is represented and how novel knowledge can be derived from prior one. Each logic comes with several *inference rules*, dictating how to produce new knowledge under particular circumstances. When coupled with some suitable *resolution strategy*, inference rules can become deterministic algorithms that computers can execute.

Many logics exist in CL – e.g. propositional, first-order (FOL), temporal, deontic, etc. –, each one targeting a specific domain. For instance, temporal logic enables reasoning about events in time, deontic logic supports reasoning about permissions/prohibitions and their circumstances, while FOL is general-purpose. Furthermore, different inference rules exist for different logics. Some are *deductive* – drawing conclusions out of premises –, some are *inductive* – looking for general rules out of several premises-conclusion examples –, while others are *abductive*—speculating on which premises caused some conclusions. Finally, when a resolution strategy exists for some inference rule, it can be translated in some software construct, and used to provide intelligent systems with automated reasoning. Software of that sort is commonly referred to as a part of the *logic programming* (LP) paradigm.

Despite the wide availability of logics, inference rules, and resolution strategies in the LP literature, only a relatively-small amount of them have been reified into actual *logic-based technologies* (LBT in short henceforth). The Prolog language [4] represents by far the most successful story here [5]. It consists of a well-defined language [6,7] coming with several implementations [8–14].

While standard implementations of Prolog target FOL via SLDNF inference rule [15,16] and depth-first resolution strategy, most implementors have extended Prolog to support other resolution strategies as well. This is the case of Prolog implementations supporting, e.g., *constraint logic programming* [17], *constraint handling rules* [18], *tabled resolution* [19], etc.

Thanks to the versatility of FOL, it is a common practice in LP to either develop LBT either *on top* of Prolog or *from scratch*. Building LBT on top of Prolog is often preferred as they automatically inherit Prolog basic mechanisms, including e.g. the capability to perform *(i)* data structures representation via logic terms, *(ii)* knowledge representation via Horn clauses, *(iii)* logic unification, *(iv)* efficient in-memory indexing of logic information, *(v)* reasoning, via a flexible inference rule, and *(vi)* meta-level programming. This is a smart strategy when LBT must be quickly bootstrapped, yet it may result in poorly-interoperable, Prolog-tailored solutions. Conversely, when Prolog capabilities are poorly-suited for some particular problem, LBT may be designed from scratch. This involves re-designing and re-implementing most LP features *ex-novo*.

In [20] Sterling states that logic *unification* is one major contribution of LP to software engineering—thus singling a specific feature out of Prolog for its value and benefits. Similarly, many aspects of LP could be useful in AI by themselves, so each contribution should be conveniently reified into individually-useable software. Accordingly, we aim at creating an *open ecosystem* for interoperable, general-purpose LP libraries, virtually supporting multiple logics, inference rules, and resolution strategies—and possibly factorising any shared aspect.

The idea of LP as a key *technology*-enabler of intelligent application was already in place decades ago. The tuProlog project [21] was proposed for this purpose. It consists of a lightweight *malleable*, object-oriented, Java-based implementation of Prolog [22] which can be used as a library for JVM projects. Despite several versions have been proposed – bringing new features, or more platforms support [23] –, and many research products have been built upon it – such as TuCSoN [24], ReSpecT [25], LPaaS [26], or Tenderfone [27], Arg2P [28], etc. –, it still consists of a *monolithic* library targetting Prolog *alone*. However, Prolog is no silver bullet for LBT, and LP should not be reduced to Prolog alone.

Accordingly, here we propose 2P-Kt: a *reboot* of the tuProlog project providing for a common technological ground for LP. Acknowledging that most mechanisms in LP have the potential to be of general value – not necessarily tailored to any specific logic, inference rule, or resolution strategy –, 2P-Kt consists of a logic-based *ecosystem* for symbolic AI, designed and implemented by taking openness, modularity, extensibility, and interoperability into account.

The tuProlog project has been completely re-designed and re-written, splitting LP functionalities into minimal, loosely-coupled, Prolog-agnostic, individually-useable, multi-platform *modules*. The rationale behind this choice is to enable the *incremental* addition of novel LP functionalities to the 2P-Kt ecosystem – possibly targeting other inference rules and search strategies –, minimising duplication of features and reusing pre-existing ones, while supporting as many programming platforms as possible. On the long run, 2P-Kt aims at becoming a comprehensive technological playground supporting several sorts of logics and inference mechanisms.

Finally, we acknowledge the importance of keeping 2P-Kt widely interoperable at the technological level with as many platforms as possible—to maximise the pool of potential adopters. Following this purpose, 2P-Kt leverages on the Kotlin multi-platform technology: each module currently supports the JVM, JS, and Android platforms—while others are expected to be supported soon.

2. Software description

2P-Kt is deeply rooted in LP, a programming paradigm based on CL [29,30]. In LP, programs are typically *theories* (a.k.a. *knowledge bases*, KB), i.e. collections of sentences in logical form, expressing *facts* and *rules* about some domain, typically in the form of *clauses*, i.e.:

$$Head \ :- \ Body_1, \ \ldots, \ Body_n$$

where both *Head* and $Body_i$ are *atomic* formulæ, and the whole sentence is read declaratively as logical implication (right-to-left). If $n = 0$, the clause is called a *fact*, a *rule* otherwise. An atomic formula is an expression in the form $P(t_1, \ldots, t_m)$ where P is a m-ary predicate ($m \geq 0$), and t_j are *terms*. Terms are the most general sort of data structure in LP languages. They can be *constant* (either *numbers* or *atoms*/strings), *variables*, or recursive elements named *structures*. Structures are used to represent clauses, lists, sets, or other sorts of expressions.

Logic *solvers* exploit KB to answer users' *queries* via some inference procedure and resolution strategy. For instance, Prolog interpreters exploit a *deductive* procedure rooted into the SLDNF resolution principle [16,31], and a depth-first strategy. Yet, other options exist like, e.g., abductive [32], inductive [33], probabilistic [34] inference. Each of them represents a particular reification of a logic solver.

A common mechanism in LP is the *unification* algorithm [35] for constructing a *most general unifier* (MGU) among any two terms. Provided that a MGU exists, its subsequent *application* to the terms, makes them syntactically equal. This is a basic brick in

virtually all LP algorithms, regardless of the particular inference rule.

Summarising, LP leverages several mechanisms – terms and clauses representation, knowledge base storage, unification, resolution, etc. –, which constitute the basis of any logic solver. Subsets of these mechanisms may be useful *per se*. 2P-Kt makes LP mechanisms *individually* available, while easing the construction of novel mechanisms on top of the existing ones.

2.1. Software architecture

Architecturally, 2P-Kt is a framework supporting LBT development via several loosely-coupled *modules*. To support reusability, each module factorises related functionalities via compact API of OOP types and methods. As modules are the basic deployable units in 2P-Kt, major LP functionalities are partitioned into modules on a per-usage basis, making them selectively useable as dependencies by other projects. The 2P-Kt ecosystem itself incrementally combines such modules, as depicted in Fig. 1.

To maximise interoperability, 2P-Kt modules are individually available as precompiled libraries both on Maven Central Repository [36] – for JVM-, Android- or Kotlin-based contexts – and on the NPM Registry [37] – for JavaScript-based contexts –, whereas a detailed description of their API is available as a part of 2P-Kt documentation.

If all 2P-Kt modules were merged together, the most relevant aspects of their API could be summarised as in Fig. 2. The diagram shows how relevant LP aspect are reified into types: e.g.

- logic `Terms` (plus any specific sort of term, e.g. `Variables`, `Structures`, etc.),
- logic `Substitutions`, unification, and MGU (computed by an `Unificator`),
- `Clauses` (there including `Rules`, `Facts`, and `Directives`),
- knowledge bases and logic `Theory`,
- automatic reasoning, via `Solvers`, and
- logic `Solutions`—i.e. responses to users' queries.

Interfaces expose relevant aspects, and keep the system extensible. Developers may for instance define custom implementations for the `Unificator` and `Solver` interfaces, to provide novel inference mechanisms involving some variant of unification.

Of course, a detailed diagram would include more features, as 2P-Kt supports: *(i)* (de)serialisation of logic terms and theories into/from standard data-representation formats (e.g. JSON, or YAML), *(ii)* parsing/formatting terms and theories from/into concrete logic syntaxes such as Prolog's one, *(iii)* extension of solvers via libraries of custom LP functionalities, and *(iv)* exploitation of solvers via command-line (CLI) or graphical (GUI) user interfaces, too.

2.2. Software functionalities

Here we enumerate 2P-Kt functionalities on a per-module basis. Following Gradle convention, we denote modules by *:moduleName*.

The most fundamental module is *:core*, which exposes data structures for knowledge representation via terms and clauses, other than methods supporting their manipulation—e.g. construction, unfolding, scoping, formatting, etc. Novel sorts of terms/clauses may be added by developers, by extending any public interface in *:core*. Furthermore, all types in *:core* leverage an *immutable* design, making them well suited for concurrent and multi-threaded scenarios.

Logic terms and clauses are often compared or manipulated via *unification*. Thus, we encapsulate this mechanism within the *:unify* module. It provides a general notion of `Unificator` – i.e. any algorithm aimed at computing MGU out of terms or clauses –, and a default implementation based on [35]. Developers may extend this implementation by configuring when terms should be considered equal. Similarly, they can provide custom `Unificator` implementations, in case they need a specific unification strategy, or need a different unification algorithm.

Another common need in LP is the *in-memory* storage of clauses into ordered (e.g. queues) or unordered (e.g. multisets) data structures, and their efficient retrieval via pattern-matching (e.g. unification). The *:theory* module follows this purpose, by providing notions such as `ClauseQueue`, `ClauseMultiset`—both involving an *immutable* (access-efficient) and *mutable* (update-efficient) implementation. These types differ from ordinary collections as they support unification-based retrieval and indexing of clauses. Prolog's notions of theory and static/dynamic KB leverage on these types, exploiting the most adequate implementation in each case.

The practice of LP also involves ancillary operations over terms and clauses, e.g.: *(i)* formatting – into some *costumisable* form –, *(ii)* (de)serialisation – into/from data-representation formats –, and *(iii)* parsing—out of a particular concrete syntax. While formatting is a *:core* functionality, attained via `TermFormatters`, (de)serialisation and parsing require their own modules. Accordingly, module *:serialize-core* (resp. *-theory*) supports the (de)serialisation of terms (resp. theories) into JSON or YAML, via human-readable schemas. Thus, it supports distributed applications exchanging logic knowledge over the Internet. Similarly, parsing terms (resp. theories) in Prolog syntax is supported through the *:parser-core* (resp. *-theory*) module, leveraging ANTLR technology [38] for language engineering.

Generic API for logic solvers are available too, via the *:solve* module. Essentially, this module exposes the `Solver` type, representing any entity capable of performing some sort of logic resolution to provide `Solutions` to logic queries. However, resolution involves many practical aspects which are *orthogonal* w.r.t. any particular resolution strategy—e.g. errors management, extensibility via libraries, I/O, etc. Thus, *:solve* is quite an articulated (yet not directly useable) module.

Developers may build their inference procedure of choice by providing an implementation for the `Solver` interface, possibly reusing features from *:solve* in a selective way. Two implementations are currently available as part of 2P-Kt, namely *:solve-classic* and *-streams*, both implementing Prolog's SLDNF resolution strategy. The latter is based on a state-machine-based design [22] and is currently stable, while *:solve-streams* is an experimental attempt of implementing Prolog via FP, as proposed in [39]. Notably, none of them leverages Warren's Abstract Machine [40]—the computational model Prolog is commonly built upon.

Generic API for developing Prolog-like predicates in Kotlin are available as well. They exploit FP and OOP to let developers extend solvers with libraries of complex functionalities which are easier to implement in Kotlin than LP. There, data streams are treated as flows of solutions to be consumed by a solver. This makes 2P-Kt well-suited for handling long/infinite streams of data [41].

User experience (UX) is enabled by two more modules, namely *:repl* and *:ide*, which provide a CLI and GUI, respectively. While they both target JVM-specific UX, an experimental web-based GUI is available at [42], targetting JS-specific UX.

Other modules depicted in Fig. 1 do not need a specific description here: interested readers may read [43] for further details.

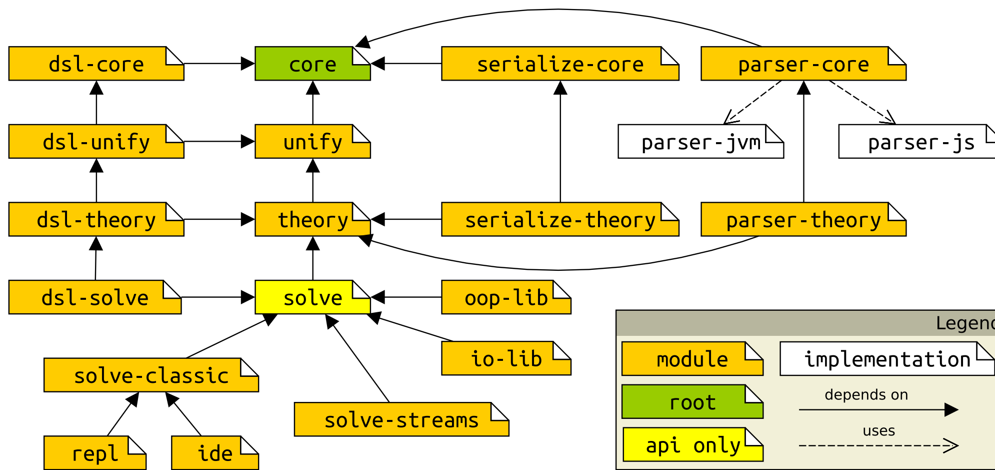


Fig. 1. 2P-Kt project map. LP functionalities are partitioned into some loosely-coupled and incrementally-dependent modules.

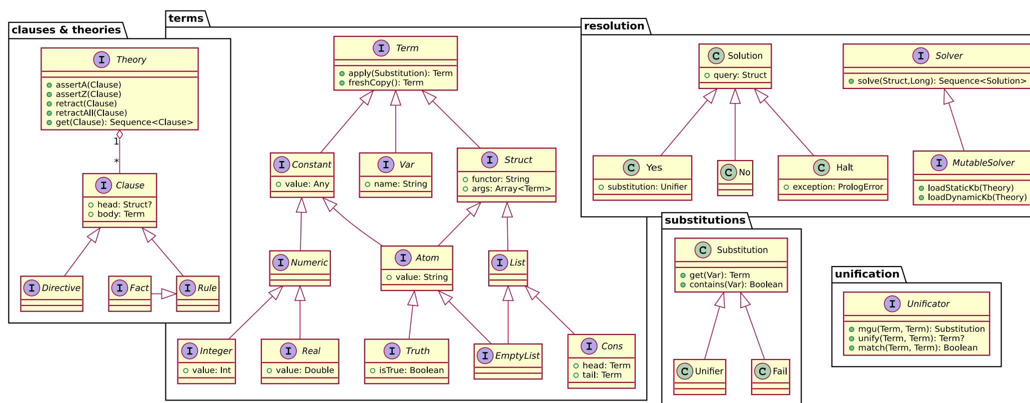


Fig. 2. 2P-Kt public API. A type is provided for each relevant concept in LP.

3. Illustrative examples

The 2P-Kt GUI (Fig. 3(a)) is a minimal IDE based on JavaFX. It lets users exploit LP interactively, repeatedly editing theories, performing queries, and inspecting the mutable internals of logic solvers. Users can open several files at once, perform queries one-by-one or all-at-once, or inspect the currently loaded libraries, operators, flags, etc. Syntax colouring completes the picture, easing users' writing of logic theories.

The CLI (Fig. 3(b)) lets users use logic solvers via a textual console. It supports both an interactive and non-interactive operation mode. Thus, it can either enter a Read-Eval-Print-Loop accepting logic queries from `stdin` and progressively prompting solutions to `stdout`, or simply accept queries and theories as arguments and prompts all possible solutions.

The Playground (Fig. 3(c)) is a proof-of-concept Web application mimicking the IDE. It demonstrates how 2P-Kt can be executed in-browser in a server-less fashion. It only requires Internet connection upon page loading. After that, it does not interact with the server anymore as the 2P-Kt JS scripts provide for a self-sufficient environment. Thus, logic computations need not any sandbox, nor logic solvers need API limitation for security reasons.

Finally, our Kotlin-based DSL for Prolog [43] can be exploited within Kotlin projects (Fig. 3(d)), by using any *`:dsl-*`* module as dependencies. It provides logic programmers with a syntactical way to inject LP into Kotlin scripts, making it possible to inherit the many tools available for Kotlin development, e.g. type checking, linting, code completion, debugging, etc.

4. Impact

The 2P-Kt technology may impact on many research areas.

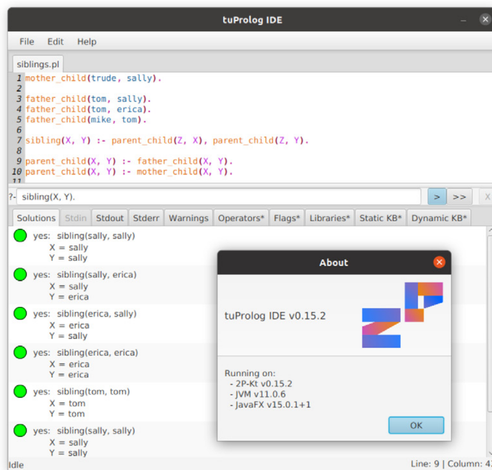
As far as LP is concerned, 2P-Kt provides a well-grounded technological basis for implementing/experimenting/extending the many solutions proposed in the literature—e.g., abductive inference [32], rule induction [33], probabilistic reasoning [34], labelled LP [44].

As shown in [3], the multi-agent systems community has quite an appetite for *interoperable* and general-purpose LBT. There, 2P-Kt provides a technological substrate supporting agents' reasoning via manifold mechanisms.

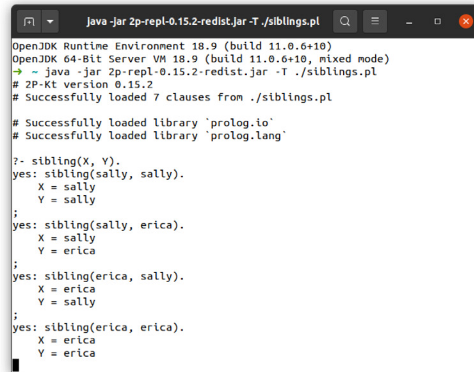
2P-Kt is a valuable choice within the field of coordination [45], too: many tuple-based coordination models and technologies leverage LP and LBT [46]. There, 2P-Kt enables the implementation of *interoperable* Linda tuple spaces – such as in TuSoW [47] – or tuple *centres*—as we plan to do in TuCSoN [24].

Concerning programming paradigms, while most successful ones are being increasingly *blended* into modern programming languages, LP remains somewhat isolated [43]. Our Kotlin DSL for LP paves the way towards the integration of LP with other paradigms.

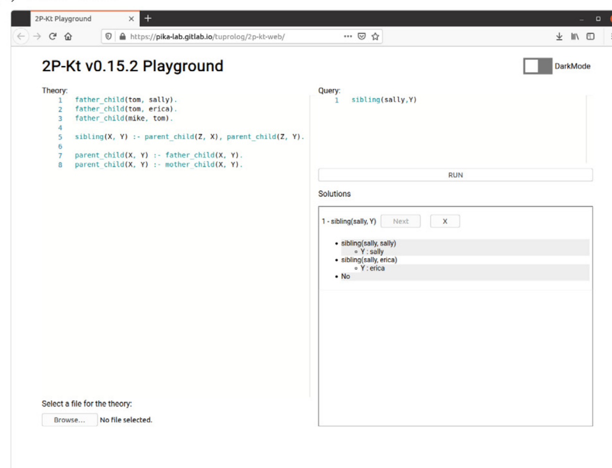
Finally, 2P-Kt has a role to play in the field of XAI [48]. Integrating symbolic and sub-symbolic AI – i.e. using them in synergy, as an ensemble [49] – is a strategical research direction [2], and 2P-Kt offers a sound technological foundation for this purpose [50].



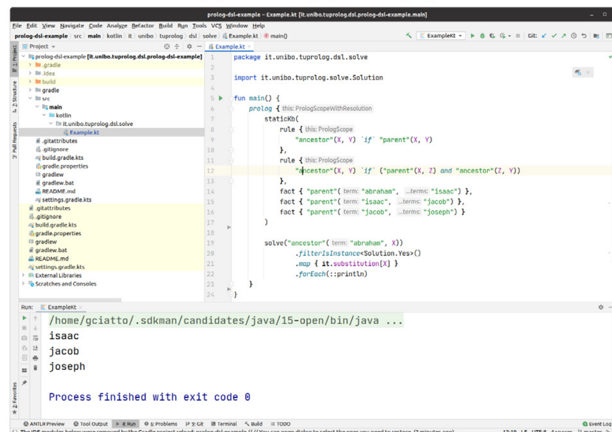
(a) The GUI



(b) The CLI



(c) The Playground [42]



(d) The Kotlin DSL for Prolog [43]

Fig. 3. Usage examples for 2P-Kt.

Future research directions. 2P-Kt already enables the investigation of relevant research questions involving symbolic manipulation or automated reasoning, thanks to its modularity and interoperability.

Furthermore, 2P-Kt enables exploring how to: *(i)* integrate different LP aspects, *(ii)* blend LP with other AI techniques, and *(iii)* exploit LP to build flexible intelligent systems. Along these lines, our goals involve: *(i)* creating comprehensive solvers exploiting multiple inference procedures, knowledge-representation means, etc. at once in answering users' queries, *(ii)* building hybrid systems where developers can transparently exploit sub-symbolic AI, and *(iii)* injecting LP into cognitive agents architectures.

As far as goal *(i)* is concerned, we are designing a unified API for probabilistic, abductive, or concurrent resolution. This would

enable further research towards *mixed* automated reasoning processes, where multiple inference procedures are dynamically interleaved within resolution.

As far as goal *(ii)* is concerned, we are designing logic-based API for machine learning and neural networks. Such API allow developers to define, train, assess, and use sub-symbolic predictors via LP. This would enable further research w.r.t. the integration of symbolic and sub-symbolic AI, the automation of machine learning workflows, and the exploitation of induced knowledge in LP.

Finally, about goal *(iii)*, we are integrating multiple logics within BDI architectures. Intelligent agents may then adopt the most adequate reasoning or knowledge-representation means for the situation at hand. Thus, 2P-Kt enables further research towards the exploitation of different logics to support intelligent, context-specific behaviours for software agents, by providing the underlying reasoning facilities.

2P-Kt adoption. While tuProlog has been exploited both in the industry and in the academia [51], 2P-Kt has been used in the academia only.

2P-Kt already works (or is going to work) as the underlying technology of many scientific contributions. Some, such as TuCSoN [24], ReSpecT [25], LPaaS [26], or Tenderfone [27] leveraged on tuProlog for their implementation, and are being migrated on 2P-Kt. Others, such as TuSoW [47], Arg2P [28], or our Kotlin-based DSL [43] already exploit 2P-Kt.

5. Conclusion

This paper introduces 2P-Kt, an open, general, Kotlin Multiplatform ecosystem for LP, supporting manifold mechanisms for automated reasoning, via several loosely-coupled modules. Each module makes some specific LP aspects individually useable. Selectively reusing/extending modules enables bootstrapping novel LBT without re-implementing everything from scratch or producing Prolog-centered monoliths. In particular, 2P-Kt supports *mixed* inference procedures, involving both symbolic and sub-symbolic techniques.

The 2P-Kt ecosystem is structured by keeping reusability, extensibility and interoperability in mind. Its functionalities include knowledge representation, (de)serialisation, parsing (and formatting facilities), unification, clause in-memory indexing and storage facilities, logic inference via SLDNF, UX, and rich Kotlin API for developers. They all support JVM, JavaScript, and Android platforms.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

R. Calegari has been supported by the H2020 ERC Project "CompuLaw" (G.A. 833647). A. Omicini has been partially supported by the CHIST-ERA IV project "EXPECTATION" (G.A. CHIST-ERA-19-XAI-005).

The authors would like to thank Enrico Siboni for his contribution in the early phases of 2P-Kt development.

References

[1] Paulson LC. Computational logic: its origins and applications. Proc R Soc Lond Ser A Math Phys Eng Sci 2018;474(2210):20170872. http://dx.doi.org/10.1098/rspa.2017.0872.

[2] Calegari R, Ciatto G, Omicini A. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. In: Baldoni M, Bergenti F, Monica S, Vizzari G, editors. Intell Artif 2020;14(1):7–32, Special issue for the twentieth edition of the workshop 'from objects to agents'. http://dx.doi.org/10.3233/IA-190036.

[3] Calegari R, Ciatto G, Mascardi V, Omicini A. Logic-based technologies for multi-agent systems: A systematic literature review. In: Bergenti F, Baldoni M, Winikoff M, Seghrouchni AEF, editors. Auton Agents Multi-Agent Syst 2021;35(1):1:1–67, Collection "Current Trends in Research on Software Agents and Agent-Based Software Development". http://dx.doi.org/10.1007/s10458-020-09478-3.

[4] Colmerauer A, Roussel P. The birth of Prolog. In: Lee JAN, Sammet JE, editors. History of programming languages conference (HOPL-II). ACM; 1993, p. 37–52. http://dx.doi.org/10.1145/154766.155362.

[5] Calegari R, Ciatto G, Denti E, Omicini A. Logic-based technologies for intelligent systems: State of the art and perspectives. In: Susilo W, editor. Information 2020;11(3):1–29, Special Issue "10th Anniversary of Information—Emerging Research Challenges". http://dx.doi.org/10.3390/info11030167.

[6] ISO/IEC JTC 1/SC 22 Technical Committee. ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core. ISO/IEC 13211-1, ISO/IEC; 1995, p. 199, URL https://www.iso.org/standard/21413.html.

[7] ISO/IEC JTC 1/SC 22 Technical Committee. ISO/IEC 13211-2:2000: Information technology — Programming languages — Prolog — Part 2: Modules. ISO/IEC 13211-2, ISO/IEC; 2000, p. 23, URL https://www.iso.org/standard/20775.html.

[8] BProlog home page. 2021, URL http://www.picat-lang.org/bprolog. [Last access: August 8, 2021].

[9] Ciao! Prolog home page. 2021, URL https://ciao-lang.org. [Last access: August 8, 2021].

[10] ECLiPSe Prolog Home page. 2021, URL https://eclipseclp.org. [Last access: August 8, 2021].

[11] SICSTus Prolog home page. 2021, URL https://sicstus.sics.se. [Last access: August 8, 2021].

[12] SWI Prolog home page. 2021, URL https://www.swi-prolog.org. Last access: August 8, 2021.

[13] τProlog home page. 2021, URL http://tau-prolog.org. [Last access: August 8, 2021].

[14] XSB Prolog home page. 2021, URL http://xsb.sourceforge.net. [Last access: August 8, 2021].

[15] Robinson J. A machine-oriented logic based on the resolution principle. J ACM 1965;12(1):23–41. http://dx.doi.org/10.1145/321250.321253.

[16] Clark KL. Negation as failure. In: Gallaire H, Minker J, editors. Logic and data bases. Boston, MA: Springer; 1978, p. 293–322. http://dx.doi.org/10.1007/978-1-4684-3384-5_11.

[17] Jaffar J, Lassez J-L. Constraint logic programming. In: 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'87). New York, NY, USA: Association for Computing Machinery; 1987, p. 111–9. http://dx.doi.org/10.1145/41625.41635.

[18] Frühwirth TW. Theory and practice of constraint handling rules. J Log Program 1998;37(1–3):95–138. http://dx.doi.org/10.1016/S0743-1066(98)10005-5.

[19] Swift T, Warren DS. XSB: Extending Prolog with tabled logic programming. Theory Pract Logic Program 2012;12(1–2):157–87. http://dx.doi.org/10.1017/S1471068411000500.

[20] Sterling L, Yal̨cinalp U. Logic programming and software engineering—implications for software design. Knowl Eng Rev 1996;11(4):333–45. http://dx.doi.org/10.1017/S026988890000802X.

[21] Denti E, Omicini A, Ricci A. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In: Ramakrishnan I, editor. Practical aspects of declarative languages. Lecture notes in computer science, vol. 1990, Springer Berlin Heidelberg; 2001, p. 184–98, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings. http://dx.doi.org/10.1007/3-540-45241-9_13.

[22] Piancastelli G, Benini A, Omicini A, Ricci A. The architecture and design of a malleable object-oriented Prolog engine. In: Wainwright RL, Haddad HM, Menezes R, Viroli M, editors. 23rd ACM symposium on applied computing (SAC 2008), vol. 1. Fortaleza, Ceará, Brazil: ACM; 2008, p. 191–7, Special Track on Programming Languages. http://dx.doi.org/10.1145/1363686.1363739.

[23] Denti E, Omicini A, Calegari R. tuProlog: Making Prolog ubiquitous. ALP Newsletter 2013. URL http://www.cs.nmsu.edu/ALP/2013/10/tuprolog-making-prolog-ubiquitous/.

[24] Omicini A, Zambonelli F. Coordination for internet application development. In: Tolksdorf R, Ciancarini P, editors. Auton Agents Multi-Agent Syst 1999;2(3):251–69, Special Issue: Coordination Mechanisms for Web Agents. http://dx.doi.org/10.1023/A:1010060322135.
[25] Omicini A, Denti E. From tuple spaces to tuple centres. Sci Comput Program 2001;41(3):277–94. http://dx.doi.org/10.1016/S0167-6423(01)00011-9.
[26] Calegari R, Ciatto G, Mariani S, Denti E, Omicini A. LPaaS as mIcro-intelligence: Enhancing IoT with symbolic reasoning. Big Data Cogn Comput 2018;2(3). http://dx.doi.org/10.3390/bdcc2030023.
[27] Ciatto G, Mariani S, Omicini A, Zambonelli F. From agents to blockchain: Stairway to integration. In: Tonelli R, Ortu M, Pinna A, editors. In: Advances in blockchain technology and applications 2020, Appl Sci In: Advances in blockchain technology and applications 2020, 2020;10(21):7460:1–22.Special Issue "Advances in Blockchain Technology and Applications 2020". http://dx.doi.org/10.3390/app10217460,
[28] Pisano G, Calegari R, Omicini A, Sartor G. Arg-tuProlog: A tuProlog-based argumentation framework. In: Calimeri F, Perri S, Zumpano E, editors. CILC 2020 – Italian conference on computational logic. Proceedings of the 35th italian conference on computational logic. CEUR workshop proceedings, vol. 2719, Aachen, Germany: Sun SITE Central Europe, RWTH Aachen University; 2020, p. 51–66, URL http://ceur-ws.org/Vol-2710/paper4.pdf.
[29] Lloyd JW, editor. Computational logic. In: Computational logic: Its origins and applicationsogic. ESPRIT basic research series, Berlin, Heidelberg: Springer; 1990, http://dx.doi.org/10.1007/978-3-642-76274-1.
[30] Metakides G, Nerode A. Principles of logic and logic programming. Studies in computer science and artificial intelligence, Amsterdam, The Netherlands: North-Holland; 1996, URL https://www.elsevier.com/books/principles-of-logic-and-logic-programming/metakides/978-0-444-81644-3.
[31] Kowalski RA. Predicate logic as programming language. In: Rosenfeld JL, editor. Information processing. Proceedings of the 6th IFIP congress. North-Holland; 1974, p. 569–74.
[32] Fung TH, Kowalski R. The IFF proof procedure for abductive logic programming. J Log Program 1997;33(2):151–65. http://dx.doi.org/10.1016/S0743-1066(97)00026-5.
[33] Muggleton S, de Raedt L. Inductive logic programming: Theory and methods. J Log Program 1994;19–20(Suppl. 1):629–79, Special Issue: Ten Years of Logic Programming. http://dx.doi.org/10.1016/0743-1066(94)90035-3.
[34] de Raedt L, Kimmig A. Probabilistic (logic) programming concepts. Mach Learn 2015;100(1):5–47. http://dx.doi.org/10.1007/s10994-015-5494-z.
[35] Martelli A, Montanari U. An efficient unification algorithm. ACM Trans Program Lang Syst 1982;4(2):258–82. http://dx.doi.org/10.1145/357162.357169.
[36] 2P-Kt. Artefacts on Maven Central Repository. 2021, URL https://search.maven.org/search?q=g:it.unibo.tuprolog. [Last access: August 8, 2021].
[37] 2P-Kt. Artefacts on NPM registry. 2021, URL https://www.npmjs.com/org/tuprolog. [Last access: August 8, 2021].
[38] Parr T. The definitive ANTLR 4 reference. 2nd ed. Pragmatic Bookshelf; 2013, URL https://www.oreilly.com/library/view/the-definitive-antlr/9781941222621/.
[39] Carlsson M. On implementing Prolog in functional programming. New Gener Comput 1984;2(4):347–59. http://dx.doi.org/10.1007/BF03037326.
[40] Warren DH. An abstract Prolog instruction set. Technical note 309, AI Center, SRI International; 1983, URL https://www.sri.com/publication/an-abstract-prolog-instruction-set/.
[41] Ciatto G, Calegari R, Omicini A. Lazy stream manipulation in Prolog via backtracking: The case of 2P-KT. In: Faber W, Friedrich G, Gebser M, Morak M, editors. Logics in artificial intelligence. Lecture Notes in Computer Science, vol. 12678, Springer; 2021, p. 407–20, 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings. http://dx.doi.org/10.1007/978-3-030-75775-5_27.
[42] 2P-Kt. Playground. 2021, URL https://pika-lab.gitlab.io/tuprolog/2p-kt-web. [Last access: August 8, 2021].
[43] Ciatto G, Calegari R, Siboni E, Denti E, Omicini A. 2P-KT: logic programming with objects & functions in Kotlin. In: Calegari R, Ciatto G, Denti E, Omicini A, Sartor G, editors. WOA 2020 – 21th workshop "from objects to agents". CEUR workshop proceedings, vol. 2706, Aachen, Germany: Sun SITE Central Europe, RWTH Aachen University; 2020, p. 219–36, 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 September 2020. Proceedings. URL http://ceur-ws.org/Vol-2706/paper14.pdf.
[44] Calegari R, Denti E, Dovier A, Omicini A. Extending logic programming with labelled variables: Model and semantics. In: Fiorentini C, Momigliano A, Pettorossi A, editors. Fund Inform 2018;161(1–2):53–74, Special Issue CILC 2016. http://dx.doi.org/10.3233/FI-2018-1695.
[45] Malone TW, Crowston K. The interdisciplinary study of coordination. ACM Comput Surv 1994;26(1):87–119. http://dx.doi.org/10.1145/174666.174668.
[46] Ciatto G, Di Marzo Serugendo G, Louvel M, Mariani S, Omicini A, Zambonelli F. Twenty years of coordination technologies: COORDINATION contribution to the state of art. In: De Nicola R, editor. J Logical Algebr Methods Program 2020;113:1–25. http://dx.doi.org/10.1016/j.jlamp.2020.100531.
[47] Ciatto G, Rizzato L, Omicini A, Mariani S. TuSoW: Tuple spaces for edge computing. In: The 28th international conference on computer communications and networks (ICCCN 2019). Valencia, Spain: IEEE; 2019, p. 1–6. http://dx.doi.org/10.1109/ICCCN.2019.8846916.
[48] Arrieta AB, Rodríguez ND, Ser JD, Bennetot A, Tabik S, Barbado A, García S, Gil-Lopez S, Molina D, Benjamins R, Chatila R, Herrera F. Explainable Artificial Intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf Fusion 2020;58:82–115. http://dx.doi.org/10.1016/j.inffus.2019.12.012.
[49] Cambria E, Li Y, Xing FZ, Poria S, Kwok K. SenticNet 6: Ensemble application of symbolic and subsymbolic AI for sentiment analysis. In: 29th ACM international conference on information & knowledge management (CIKM'20). New York, NY, USA: Association for Computing Machinery; 2020, p. 105–14. http://dx.doi.org/10.1145/3340531.3412003.
[50] Pisano G, Ciatto G, Calegari R, Omicini A. Neuro-symbolic computation for XAI: Towards a unified model. In: Calegari R, Ciatto G, Denti E, Omicini A, Sartor G, editors. WOA 2020 – 21th Workshop "from Objects To Agents". CEUR workshop proceedings, 2706, Aachen, Germany: Sun SITE Central Europe, RWTH Aachen University; 2020, p. 101–17, 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 September 2020. Proceedings. URL http://ceur-ws.org/Vol-2706/paper18.pdf.
[51] tuProlog. Users. 2021, URL http://apice.unibo.it/xwiki/bin/view/Tuprolog/Users. [Last access: August 8, 2021].