Künstliche Intelligenz manuscript No.
(will be inserted by the editor)

What Language Do You Use To Create Your AI Programs and Why?

Received: date / Accepted: date

1 Introduction

When 'AI Languages' was proposed as a special topic for this journal, a lively discussion started between the editors. On the one hand, several of us voiced the opinion that language development is no longer a topic of AI research and that AI researchers, just as everybody else in computer science, use some more or less mainstream language which is convenient for a given area of research. On the other hand, it became immediately clear that everybody has strong feelings or at least a qualified opinion about the languages which he or she uses to create AI programs. Because we all are interested in the question which language has what advantages in the context of our area of research, we are hoping that the readers of this journal are also interested in this topic. To get a discussion started, we asked several colleagues for a statement. The result is the collection presented in the following.

From the six collected opinions, it becomes obvious that for many current areas of research C++ or Python are good, pragmatic choices. However, there is something to say about the good, old-fashioned AI languages. In one contribution it is shown how parsimoniously and transparent a rather complex graph problem can be represented in Prolog and how efficient the problem can be solved. A further contribution shows that there is still demand for reflection and the ability of code which modifies itself – which was one of the main features incorporated in Lisp and considered crucial for an AI programming language.[1] In this contribution, the term rewriting language Maude is favoured. This language – as well as many other recent and not so recent developments in the context of functional programming – can be seen as offsprings of the basic concepts of AI programming introduced by John McCarthy. McCarthy – father of Lisp, father of AI – died at age 84 in October 2011. His work, including the introduction of abstract syntax and proof techniques for properties of programs, will live on.

2 Do You Know What You Are Doing? – Start High-Level, Then Go Mainstream

Stefan Mandl
Universität Augsburg
Lehrstuhl für Datenbanken und Informationssysteme
E-mail: mandl@informatik.uni-augsburg.de

"Computers are like a bicycle for our minds" — if this famous slogan which was propagated by recently deceased Steve Jobs is in fact true, if computers really help us think, and more specifically, if *programming* helps us think, then programming languages are the languages we have to think in when we think with the help of a computer. So here I take the liberty to rephrase the original question *"What language do you use to create your AI programs and why?"* to *"What (programming!) language do you use to think about AI systems?"* As researcher I work in two modes: engineering mode and high-level mode.

In engineering mode the system behavior to achieve is quite clearly specified and aspects such as *correctness*,

This collection of contributions was compiled and introduced by Ute Schmid, member of the editorial board.

Address(es) of author(s) should be given

[1] see, e.g., slides of Aaron Sloman's talk to first year AI students `http://www.cs.bham.ac.uk/~axs/misc/talks/setools-ailanguages.pdf`.

reliability, and *performance* are central when thinking about AI systems. Consequently, in engineering mode I use tools and languages that any engineer of any field of Computer Science would use: C(++), Java, SQL, IDEs, ..., and most important: *libraries!*

In high-level mode on the other hand, the behavior to achieve often is only a vague idea, sometimes in the form of a list of 'what-if' questions. Then aspects like *fast prototyping*, *high-level abstractions*, *meta level programming* and *conciseness* are important. The shorter the program the better, as short programs are easier to modify. In such situations I often use LISP or PROLOG for the first shot of the system. Both LISP and PROLOG share the property of being homoiconic languages (Programs are written in the form of a data structure the language supports well; i.e. lists in LISP and first-order terms in PROLOG), which allows for stunning meta-programming facilities right out of the box. Modest datasets can be represented and manipulated directly in source code alongside the algorithms and by using features such as the very powerful LISP macros, the programming language literally can be tailored to the problem at hand. Ideally a domain specific language would emerge that would be well suited to implement the desired behavior. Sadly enough we do not live in an ideal world and henceforth, as the program matures, at least in my projects, LISP and PROLOG tend to show their weaknesses. As LISP and PROLOG are rarely used these days, it is difficult to share the program with colleagues. When the system is developed in the context of an industrial research project, industrial partners usually have very specific constraints regarding the implementation language. But even when building research-only prototypes, sooner or later one usually misses a certain library or framework (*"the demo looks great now, so let's run it on our Tomcat server"*). For me this is the time to think about re-implementation in a more mainstream programming language. One could regard this as a waste of time, but I found that usually re-implementation is not a bad idea for a prototype that has been created in a hurry.

Let me illustrate this approach by the following example. For my thesis about handling unconsidered contexts of formalized knowledge [1], initially I had a lot of different ideas involving Genetic Algorithms, Machine Learning, and Logic Programming. The first prototype implementation was created using LISP. LISP's powerful object system (CLOS) allowed me to write very general algorithms that worked across different domains like planning, and answer set programming. Soon enough I had developed a bunch of macros for setting up and integrating new problem domains. Sure some external tools had to be integrated by using implementation specific constructs for hooking up to external operating system processes, but I got quite far. Until I reached the point where I wanted to create a demo for coping with unconsidered context in an image classification scenario. There, the use of external tools became more and more painful, until I finally decided that it is time to re-implement the algorithms in JAVA. Starting an implementation from a working prototype is much easier than starting from scratch and most importantly, I had already figured out all the algorithms and which data structures I wanted to use in the new version. The re-implementation only took a few weeks (part time, as I had other duties as well), and to this day I never looked at the LISP version again.

So starting at a high level and then going mainstream turned up to be the right approach for me. I'm in no position to recommend this approach to anyone else. Probably one could also think a lot first, work it all out on paper and then implement it, or one's mind is simply tuned to think in JAVA. But for me, as I want the computer as a bicycle for *my* mind, starting high-level and then going mainstream is the way to go when building systems I initially do not fully understand.

References

1. Mandl, Stefan: *Erschließung des unberücksichtigten Kontexts formalisierten Wissens.* In: Künstliche Intelligenz 23 (2009), Nr. 1, S. 60-62

3 The Zest for Prolog Programming

Helmar Gust
Institute for Cognitie Science
University of Osnabrück
E-mail: hgust@uos.de

3.1 Introduction

When invented in the early seventies Prolog was a real breakthrough in programming paradigms. It was conceptually developed by Alain Colmerauer built on a logical basis suggested by Robert Kowalski followed by a first implementation by Philippe Roussel [1]. The next milestone was David Warren's method of compiling logic programs into native code of a virtual machine [2]. Moreover, this work established the standard Prolog syntax used in the legendary DEC 10 implementation, which essentially is used until today. Based on the 1965 advancement in automatic theorem proving technology [3] Prolog uses very basic but powerful principles: unification and a very simple version of resolution. This

ensures that Prolog programs have a logical and a procedural interpretation.

This together with the fact that the core of Prolog can be implemented in a very efficient way enables the usage of Prolog as a general purpose programming language, although it was originally invented for a quite restricted field of application: The implementation of grammars.

In the early eighties the Fifth Generation Computer Systems (FGCS) initiative of the MITI (Japanese Ministry of International Trade and Industry) pushed the logic programming paradigm and tried to develop special hardware that runs logic programs as native code [4]. The goal to establish a new computer hardware/-software paradigm failed mainly for two reasons:

1. Progress in general purpose hardware was so fast that it easily outperformed all types of specialized hardware like the FGCS developments, but also the Lisp machines.
2. The choice for concurrent logic programming [5] with committed choice turned out to be not a clean enough concept. Progress in constraint based programming opened a different direction of giving logic programming a cleaner basis.

Constraint Logic Programing had become a popular issue. This is not restricted to Prolog or extending Prolog with solvers for constraint satisfaction problems (CSPs). Solving CSPs has become an own field of research and there are many approaches based on different host languages [6].

3.2 Essential Prolog Features

There are still a lot of groups working with Prolog, but in the application domain Prolog seems to live only in niches (e.g. Watson uses Prolog for NLP [7]).

I think that nowadays Prolog is underestimated. The essential features that fascinated me from the beginning in developing programs in Prolog are:

1. Logical variables: do not bother about variable assignment, Prolog will do it for you (except inside negations).
2. Weak typing: don't bother about types, every type of data may occur everywhere.
3. Flat and modular program structures: every clause has its own semantics (except in the context of cut).
4. Few and transparent evaluation principles (like unification and backtracking search).
5. Clean interface to add additional functionality like solvers for different CSP domains.
6. Weak discrimination between data and code: build data and evaluate it as code without extra transformations.
7. incremental compilation and decompilation (*assert* and *retract* clauses dynamically).

3.3 Some Experiences with Prolog

Of course, the relation of a programmer to a programming language is largely determined by the individual history and experience. So I would like to present here two experiences of mine that somehow shed some light on the reasons for my zest for programming in Prolog.

I remember my first contact with Prolog trying out Micro Prolog[2] on a CPM machine. For those who are too young to have come across such a computer: less than 64 KB of memory, a 1MHz 8 Bit processor, only floppies as mass storage. Micro Prolog reached not much more than 100 inferences per second on such a machine. We tried reimplementing a natural language question answering application (we developed in Simula[3] on the IBM VM/370 mainframe of the computer center; some thousand lines of code).

We were deeply impressed by the compactness of the Prolog code the performance of the program and the ease of reimplementing the basic functionality of our system: for tasks that run for minutes on the mainframe the Micro Prolog program reacted nearly immediately. We had the impression that if we had used Prolog from the beginning on the project we would have saved dozens of month of hard programming work. To get familiar with this strange new programming paradigm I deconstructed Micro Prolog and implemented my own extended Prolog system.

Another more general and very typical situation I faced several times is the following: A student has to write a Bachelor thesis and wants to use some standard AI methods in an application field, e.g. planning. The student is familiar with Java, since they learn Java in the first semester and with Prolog later they do not really get familiar. At the end she comes up with an implementation of a planning algorithm in Java and some little application: all together about a thousand lines of code. The program works more or less well, but the student dramatically underestimated the implementation effort, because she had to implement all the data-structures and the unification procedure and she had to handle non-determinism by her own. Moreover, the program was flawed and couldn't handle a lot

[2] a small Prolog using a Lisp like syntax. Some times I yearn for such a minimalistic language.

[3] One of the first object oriented programming languages

of cases correctly. My remark "in Prolog it would have been thirty lines of code and it would have worked" was of course quite frustrating for her. Whenever one needs unification and/or nondeterminism it is quite hard to implement this from the scratch.

3.4 A Challenging Example

As a small but challenging example the reader might have a look at the following problem computing hamiltonian paths in graphs wrapped in the following task[4]: "Write a program that computes all possibilities to run a cooling duct through a computer center touching all rooms which belong to the computer center exactly once, respecting the give intakes and outlets. Rooms are organized in a grid and there are rooms which do not belong to the computer center." A Prolog solution can be found in [8]. It nicely shows how modular and well structured a solution can be. Nearly all the code in simply specifying the problem. The solver itself is four lines of trivial code[5]. It uses nearly all the features of Prolog listed in the above list. The solution process is quite efficient. It runs in 25 seconds for the original problem on Eclipse Prolog on my MacBook Pro[6]. I wonder if there is any implementation in another programming language that allows such an abstract specification of the problem and the solver and that turns out to be comparable efficient. Further more, what is quite striking: The solution is even much more general than requested by the requirements, since it is able so solve the problem without any change in case of multiple intakes and outlets.

3.5 Conclusion

The classical competitor having some of the mentioned features is, of course, Lisp, but in Lisp there are no logical variables, no unification, no non-determinism. Function definitions tend to be mountains of parantheses. Of course, there are modern alternatives like Python (weak typing and nice data structures), ML and Haskell (type inferences), and PHP (something like weakly typed C for Web interfaces), and I use all of these in special cases. But if it comes to attack a new problem that is not strongly related to building a user interface, then my first try is Prolog.

There are a lot of things I have to criticizes in the standard Prolog implementations, things where life could be made easier in writing Prolog programs.

- The intermixture of procedural and logical semantics of clauses has some shortcomings (purely declarative code can sometimes be evaluated more efficiently while losing the procedural interpretation).
- Need for more flexible standard input (it is quite boring to write complex code to simple read a natural language sentence).
- Lack of a functional sub-language (although there are approaches to compile functional code to Prolog [9]).

Nevertheless, for people like me who somehow internalized the logic programming paradigm solving problems practically means writing Prolog programs.

References

1. A. Colmerauer, P. Roussel, in *III, CACM Vol.33, No7* (1993), pp. 37–52
2. D.H. Warren, An abstract prolog instruction set. Technical note, Artificial Intelligence Center @ SRI (1983)
3. J.A. Robinson, Journal of the Association for Computing Machinery **12(1)**, 2341 (1965)
4. P. Rouchy, TeamEthno - Online Issue **2**, 85 (2006). June 2006
5. E. Shapiro, ACM Computing Surveys **21(3)**, 412510. (1989)
6. F. Rossi, P. van Beek, T. Walsh (eds.). *Handbook of Constraint Programming* (Elsevier, 2006)
7. A. Lally, P. Fodor. Natural language processing with prolog in the ibm watson system (2011). URL `http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-` URL `watson-system/`
8. H. Gust. Solution of the data center cooling promblem in polog (2011). URL `https://cogsci.uni-osnabrueck.de/~hgust/programs/quora_datacenter_cooling.pl`
9. J. Ireson-Paine (2003). URL `http://www.j-paine.org/grips.html`

4 Code as Data – Creating Programs Which Create Programs With Maude

Emanuel Kitzelmann[7]
International Computer Science Institute, Berkeley, USA
E-mail: emanuel@icsi.berkeley.edu

Before I started to implement my inductive programming algorithm IGOR 2 [2], which I had developed as part of my doctoral research, I had used LISP to implement inductive programming (IP) systems. LISP was a reasonable choice since IP is all about dealing with

[4] The original problem can be found at http://www.quora.com/challenges

[5] which with tricks can be even collapsed to one call

[6] roughly a factor of 10 to 20 compared to the C^{++} hand coded 'optimal' solution mentioned at http://www.quora.com/challenges

[7] The author is funded within the DAAD FIT-programme.

code as data. In particular, IP is concerned with synthesizing programs from incomplete specifications such as input/output examples or computation traces. So why did I choose the term-rewriting based language Maude [1] to implement Igor 2?

Igor 2 synthesizes recursive *constructor (term rewriting) systems (CSs)* from non-recursive CSs that specify the desired functions on parts of their domains. In the simplest case, the specifying CS consists of a number of ground rules and denotes a set of I/O examples. CSs can be seen as first-order functional programs: They consist of equations over algebraic types which are interpreted as reduction rules and may contain constructors in their heads (pattern matching). Fig. 1 shows an example of a specification and the synthesized function.

```
sort NList .  ***sort/type for lists of natural numbers
op nil : → NList [ctor] .              ***empty list
op _:_ : Nat NList → NList [ctor] .    ***cons
vars x, y, z : Nat .  var xs : NList .  *** variables

op last : NList → Nat .   ***signature for last

*** specification                ***synthesized solution
eq last(x:nil) = x .             eq last(x:nil) = x .
eq last(x:y:nil) = y .           eq last(x:y:xs) = last(y:xs) .
eq last(x:y:z:nil) = z .
```

Fig. 1 Example last: Type and specification (input to Igor 2) and induced solution (Igor 2 output) in Maude syntax.

I had two (plus one) reasons for choosing Maude. First, Maude's so-called functional modules, a certain subset of the language, are an extended form of CSs. Hence Igor 2's objects – specifications and generated programs – are valid Maude programs and I didn't need to care about implementing my own object language. Second, Maude has powerful reflection capabilities that facilitate parsing, manipulation and evaluation of Maude code from within Maude programs, just like quoted expressions in Lisp can represent code, can be manipulated by list functions and evaluated. (And third: I felt like trying something new.)

Reflection means that for all constructs of Maude programs (signatures, terms, equations, complete modules) data structures to represent and manipulate them are implemented in Maude's standard library. Meta-represented terms, equations, modules etc. are *terms* of types Term, Equation, Module etc. and can be rewritten by a Maude program just like any other term. For example, consider a Maude module, let's say a module M that contains the two equations of the synthesized solution from Fig. 1. Applying upEqs('M, false) would then yield

```
eq 'last['_:_['x:Nat,'nil.NatList]] = 'x:Nat [none] .
eq 'last['_:_['x:Nat,'_:_['y:Nat,'xs:NatList]]] =
     'last['_:_['y:Nat,'xs:NatList]] [none] .
```

which is a term of type EquationSet. Also rewriting and related concepts like matching and substitutions are implemented at the meta-level. For example,

```
metaMatch(upModule('M,false), '_:_['x:Nat,'xs:NatList],
          '_:_['1.Nat, '_:_['2.Nat,'nil.NatList]], nil, 0)
```

returns the term (of type Substitution):

```
'x:Nat ← '1.Nat , 'xs:NatList ← '_:_['2.Nat,'nil.NatList] .
```

Two further features that distinguish Maude from other (functional) programming languages are subtypes and operator properties like associativity, commutativity etc. which (together with pattern matching) permit succinct definitions of data structures. Figure 2 shows examples for lists and sets.

```
sort NatC .              ***a sort for Nat collections
subsort Nat < NatC . ***a Nat is a Nat collection, size 1
op none : → NatC [ctor] .  ***the empty collection
***we define a constructor _,_ to build collections from
*** existing ones and make it associative and having none
***as id element; the collection now corresponds to a list
op _,_ : NatC NatC → NatC [ctor assoc id: none] .
var x : Nat .   var xs : NatC .  *** variables
***now getting the last element from a list is just
***pattern matching (like getting its first element)
op last : NatC → Nat .      eq last((xs,x)) = x .
reduce last ((1,2,3)) .  Result: 3  ***a quick test
***let's make the collection a set by adding commutativity
***and eliminating multiple instances of the same element
op _,_ : NatC NatC → NatC [ctor assoc comm id: none] .
eq ((x , x)) = x .
reduce (1,2,3,2) .  Result: (1,2,3)
***every element can be the first one due to commutativity
op member : Nat NatC → Bool .
eq member (x, (x , xs)) = true .
eq member (x, xs)       = false [owise] .
reduce member(3, (1,2,3,4)).  Result: true.
```

Fig. 2 Succinctly defining data structures in Maude.

I covered some features that let me chose Maude for implementing Igor 2, but Maude has much more to offer. Types can be parameterized and besides *functional modules* there are so-called *system modules*, that let you specify and implement concurrent and non-deterministic systems, and even *object-oriented modules*. Maude is a logical framework in which more specific languages can be modeled. It is a strictly declarative language and includes a model checker such that properties of a Maude program/theory can automatically be checked. The weak points of Maude are a rather small library with only few predefined data structures and the lack of suitable input/output handling.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Springer-Verlag (2007)
2. Kitzelmann, E.: A combined analytical and search-based approach for the inductive synthesis of functional programs. Künstliche Intelligenz **25**(2), 179–182 (2011)

5 Optimize Runtime, but Also Your Own Time

Malte Helmert
Universität Basel
Fachbereich Informatik
E-mail: malte.helmert@unibas.ch

When Silvia Richter and I started working on the Fast Downward planning system [1] in 2003 with the goal of submitting it to the 4th International Planning Competition, we faced two major challenges:

1. Writing fast code: the planner had to be efficient to have a chance at winning the competition.
2. Writing code fast: the planner had to be developed quickly to meet the submission deadline.

These are conflicting goals, as efficient implementations take time: programming languages offering the best potential performance tend to work at lower levels of abstraction; maximum performance often requires specifically tailored data structures; and optimization often hurts modularity of code.

However, not all parts of a planning system – or indeed the vast majority of programs – are equally time-critical. A few operations, such as computation of heuristic values, typically require more than 90% of the computation time, yet make up less than 10% of the code. Therefore, our solution for the two challenges above was to write fast code where it mattered, and to write code fast where we could get away with it. For us, this meant using a mix of programming languages: C++ where performance was critical and Python where programmer time was the more important metric to minimize.

In 2004, the planner consisted of about 3K lines of Python code and 5K lines of C++ code. Studies and our own experience show that Python code is usually around a factor of 4 more compact than C++ code that implements identical functionality, which means that roughly 70% of the planner functionality was implemented in Python. Since then, we have integrated many new search algorithms, heuristics and other features into Fast Downward and the balance between the two programming languages has changed somewhat, but we have never regretted the basic decision of using C++ for program efficiency and Python for programmer efficiency.

In this day and age, the use of C++ to build efficient AI systems probably does not need much explanation, so I only briefly point out that for our purposes, memory efficiency is as important as runtime efficiency; search code can very quickly fill up gigabytes of RAM. Memory efficiency is one of the areas in which C++ shines compared to alternatives like Java.

The use of Python might need a bit more justification since it is still often considered as a "scripting language" to serve as glue between programs written in more traditional programming languages that perform the heavy algorithmic lifting. I believe that this is a mistaken view. Large applications are now routinely being written in modern dynamic object-oriented programming languages like Python and Ruby.[8] The defining characteristic of these programming languages is not that they can be used for "scripting" but that they work at a very high level of abstraction. Python really shines at writing algorithmic code, and in addition to using it for code where constant-factor efficiency is less important [3], I now routinely use it for proof-of-concept implementations of performance-critical code [4, 2]. We have also used Python very successfully for teaching AI, for example in a general AI practical that covered a diverse range of topics and in a planning practical where a group of students developed a complete domain-independent planning system from scratch over a semester.[9]

In summary, my message is: use the right tool for the right purpose. Efficiency matters, but not everything needs to be equally efficient. Do not be afraid to mix and match. Above all, conserve the most valuable resource at your disposal: your time.

References

1. Helmert, M.: The Fast Downward planning system. JAIR **26**, 191–246 (2006)
2. Helmert, M., Domshlak, C.: Landmarks, critical paths and abstractions: What's the difference anyway? In: Proc. ICAPS 2009, pp. 162–169 (2009)
3. Helmert, M., Röger, G., Karpas, E.: Fast Downward Stone Soup: A baseline for building planner portfolios. In: ICAPS 2011 Workshop on Planning and Learning, pp. 28–35 (2011)

[8] Indeed, almost the complete development tool-chain surrounding our planning system is implemented in Python, including the revision control system we use (Mercurial), the issue tracker (Roundup), wiki engine/website (MoinMoin) and build automation tool (BuildBot).

[9] The resulting *pyperplan* planner is online at `https://bitbucket.org/malte/pyperplan`.

4. Richter, S., Helmert, M., Gretton, C.: A stochastic local search approach to vertex cover. In: Proc. KI 2007, pp. 412–426 (2007)

6 Why We Develop Intelligent Conversational Agents with Python

Hendrik Buschmeier, Ramin Yaghoubzadeh, Christian Pietsch, Stefan Kopp
AG Sociable Agents
CITEC, Technische Fakultät, Universität Bielefeld
E-mail: hbuschme@TechFak.Uni-Bielefeld.DE

The Sociable Agents Group at the Center of Excellence "Cognitive Interaction Technology" (CITEC) at Bielefeld University aims to develop technical systems that can join humans naturally. This calls for intuitive, socially apt human-machine interaction and we explore how virtual humans and humanoid robots can be equipped with the necessary "interaction intelligence". Our research requires to understand intelligent behaviour in communication, to devise models of the underlying cognitive processes, to realise them in implemented systems and to evaluate them in actual interactions with human users. While a lot of groundwork code for behaviour animation or graphics is written in C/C++, we have increasingly come to use Python in our research on higher-level A.I. modules.

Python is an interpreted, multi-paradigm language combining procedural, object-oriented and functional aspects with transparent syntax and semantics. It is dynamically typed and has powerful meta-programming capabilities permitting changes to the program even at runtime. At the same time, Python code is easily readable, as it often resembles a description of an algorithm in pseudo-code. These properties facilitate rapid prototyping and testing, with only little structural changes needed to a minimal prototype in order to render a model hypothesis into a fully-fledged working system. This supports our research methodology of iteratively modelling and evaluating hypotheses on cognitive processes in human communication using virtual agents, which may comprise declarative and procedural knowledge structures (e.g., in the case of a dialogue manager or a reasoning strategy).

A further reason for using Python is that it comes with an extensive standard library, while powerful third-party libraries for a wide range of relevant A.I. methods are available as well. For example, machine learning and scientific computing frameworks such as SciPy or NumPy, probabilistic reasoning engines like ProbCog, or natural language processing toolkits like NLTK are available. Indeed, Python has become very popular among computational linguists because NLTK provides well-documented, mature implementations of rule-based as well as machine learning-based techniques for natural language processing [1].

Finally, Python has advantageous features from the software-engineering point of view. Conversational agents are comprehensive cognitive systems with perception, behaviour, reasoning, emotion, attention, language, or knowledge components. Technically, they are developed as distributed systems. Python supports this through availability for different platforms, inter-operability with other languages, and scalability. At the same time, modern approaches to embodied interactive systems pose high demands for close integration of perception and action, for incremental processing, or fine synchronisation of multiple processes, e.g., for different modalities. We have found Python highly useful for implementing middleware layers that provide distributed, incremental processing and different levels of integration between system components.

References

1. Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly, Sebastopol, CA, 2009.
2. Dennis Merritt. Python for AI and logic programming. Dr. Dobb's AI Expert Newsletter, August 2005, 2005.
3. Peter Norvig. Python for Lisp programmers, 2009. `http://norvig.com/python-lisp.html`

7 Why We Build Robot Control Systems in ... well, in What?

Joachim Hertzberg, Jochen Sprickerhof, Thomas Wiemann
Institut für Informatik
Universität Osnabrück
E-mail: joachim.hertzberg@uos.de

In terms of research, we are working in AI. In particular, we are working on building control software for embedded knowledge-based systems, which some would call autonomous mobile robots, some cognitive robots. To program our share of their control software, we use mainly C++.

Has the decision about the programming language been difficult? No, not really. Has it been been important? Not even that, telling in retrospect. It was purely pragmatic, and, we would say, it was also largely uninteresting. If you wish to advance the state of the art in that type of AI systems, or robots, you should better not attempt to start designing their software with

a blank page and some compiler or interpreter of whatever programming language. The reason is, again, pragmatic: To get a cognitive robot to work, you would need, in addition to an appropriate hardware platform with sensors and actuators, an amazing set of functionality. It ranges from very low-level stuff like device drivers, over components like math libraries and sensor processing libraries (which are definitely not low-level, yet out of the scope of our own research), to state-of-the art tools in knowledge representation and reasoning. Luckily, there is a huge set of open-source robotic middleware and software platforms like Player or, more recently, ROS or Orocos, which emerged much later than our original decision about a programming language, but which enhance tremendously the productivity in building research software prototypes. To decide about which of these building blocks get used, is an important, and sometimes tricky decision – much more important than the decision about the programming language in which we would add our own contributions.

Having said that, we have made no point pro any "AI programming language", but we have not made a point pro C++ or pro any other language either. Building robot control systems with the limited resources that a research team in an academic setting has, means to integrate a large variety of existing and available state-of-the-art software components, be they components with a typical AI background, or any other. The key for the success of a framework like ROS is that it provides clearly defined interfaces to combine different modules for different tasks. The question is, what task will be solved by a certain component in an integrated robot control system? How do I interface it? Once this has been defined, it can be implemented in any programming language. The additional effort that would result from taking the decision pro this or that language for programming the own contributions, in addition to the re-used software components, is relatively minor.

If you wish to contribute actively to the above-mentioned large pool of open-source software available for robot control, you better use a language that is common in the scene – and, sorry to say to an AI audience, that is primarily the robotics scene. That gives C++ a slight pro, but we would never make a principled argument out of it. On the other hand, C++ has its known nuisances and pitfalls as a programming language; we would go as far as saying that using in 2011 a programming language that lets you even think about storage allocation is downright anachronistic. Yet, at the time, more than ten years ago, when the decision had to be made about the main language to program our robot control systems, the pragmatic reasons pro C++, including the availability of able programmers in that language, were dominant, and we have never regretted the decision.

Was that an argument against specific "AI programming languages"? In our share of AI programming, we see no need to use one of them. However, our eclecticism about integrating components in our robot controllers has included from time to time to build and interface to the rest of the software little modules in Prolog, when pragmatism so suggested. In brief: The choice of a concrete programming language seems of so minor importance in our part of AI research today that we would not argue pro or contra any specific modern language, whatever its labels.

8 Conclusions

The above collection of statements highlights some (typical?) preferences for programming languages currently used to create AI programs. As we see, the days of Usenet flame wars are over and pragmatism rules over passion. For sure, there are researchers using and developping other interesting languages not covered in this discussion – among them constraint programming languages such as CHIP, query languages for the semantic web, and functional languages such as Haskell. Another aspect of interest might be which programming concepts and languages are or should be taught in AI. It would be great to continue this discussion and we strongly encourage further contributions!