LogicObjects: A Portable and Extensible Approach for Linguistic Symbiosis between an Object-Oriented and a Logic Programming Language

Sergio Castro
ICTEAM Institute
Université catholique de Louvain, Belgium
(*e-mail:* sergio.castro@uclouvain.be)

submitted 12 May 2013; accepted 5 June 2013

Abstract

Logic languages are well suited for declaratively solving computational problems that require knowledge representation and reasoning. Object-oriented programming languages benefit from mature software ecosystems featuring rich libraries and developer tools. Several interoperability solutions exist that allow a software system to be decomposed into a combination of modules implemented in a logic and an object-oriented language. Unfortunately, significative amounts of boilerplate code must routinely be written to accomplish the required interoperability. In addition, these approaches often are not amenable to custom context-dependent reification of objects in the logic world and custom mappings of arbitrary logic terms to objects in the object-oriented world. Furthermore, in the specific context of Prolog-Java interoperability, existing solutions are often compatible with only a single or a restricted set of Prolog engines and thus suffer from portability issues. To address these problems, we introduce a portable framework, relying on linguistic symbiosis, for transparently and (semi-)automatically enabling communication between routines in these two worlds, as well as a simple mechanism for customising how foreign artefacts should be reified in the other language. We validate our approach with a case study that requires a seamless integration of declarative programs in Prolog with libraries belonging to the Java ecosystem.

KEYWORDS: Logic Programming, Object-Oriented Programming, Multi-Paradigm Programming, Linguistic Symbiosis

1 Introduction and Problem Description

Logic languages are convenient for reasoning over problems of declarative nature, such as expert systems (D'Hondt, Maja and Gybels, Kris and Jonckers, Viviane 2004; Russel and Norvig 1995). On the other hand, object-oriented languages have demonstrated their usefulness for modelling a wide range of concepts found in most applications (e.g. user interfaces (Meyer 1988)). The availability of continuously growing software ecosystems around object-oriented languages, including advanced IDEs and rich sets of libraries, has also significantly contributed to their success.

Non-trivial applications can profit from implementing their components in the language that is more appropriate for expressing them. With this in mind, several approaches were developed to allow interoperability between a logic language and an object-oriented language. Most of these approaches, however, are not straightforward to implement, often requiring a significant amount of boilerplate code that obscures the real purpose of the application components. They also provide limited support for customising how to reify the foreign language objects and present compatibility issues that limit the scenarios in which they can be employed.

Linguistic symbiosis (Ichisugi et al. 1992) has been used in the past to solve the problem of seamless integration of modules written in different languages (D'Hondt, Maja and Gybels, Kris and Jonckers, Viviane 2004). By means of linguistic symbiosis, a program can invoke transparently routines defined in another language as if they were defined in its own language (Gybels 2003). Objects from one language must also be conveniently represented in the other (Wuyts and Ducasse 2001) so that a programmer can interact with them as with any other object in the current language.

We describe an approach based on linguistic symbiosis to overcome most of the current issues hampering the agile development of systems composed of object-oriented (Java) and logic (Prolog) components. Our approach allows for a fine-level control over how Java objects are reified as Prolog terms and how Prolog terms are expressed as Java objects. Our solution is not coupled to a particular Prolog implementation. Instead, we follow an *abstraction approach* as described in (Wielemaker and Costa 2011). Therefore, our libraries interact with an abstract representation of a Prolog virtual machine. This virtual machine abstraction can easily operate on any concrete Prolog engine by means of drivers. In this way, we empower programmers working on hybrid Prolog-Java systems to choose the most appropriate Prolog engine according to the requirements of a problem (*e.g.* availability of libraries, performance, or operating-system compatibility).

2 Related Work

Several techniques have been applied in existing solutions to integrate logic and object-oriented programs. In the context of Java-Prolog interoperability, the *JPL* library (Singleton et al. 2004) allows Java programmers to execute Prolog queries either from their textual representation, or using a more structured reification of Prolog terms. In addition, Java objects can be passed to a Prolog routine by means of a simple idiom. From the Prolog perspective, Java objects can be created by means of the `jpl_new/3` predicate. Other predicates are available for sending messages to Java objects and obtaining their returned results. A disadvantage of JPL, however, is that it often requires writing significant amounts of boilerplate Java code (Castro et al. 2013).

InterProlog (Calejo 2004) provides facilities similar to JPL. It allows Java programmers to pass objects to Prolog either as a reference (as in JPL) or as serialised streams of bytes. As with the JPL, the programmer is required to write quite some boilerplate code to interact with a Prolog engine from within Java, or to make use of special predicates for interacting with Java objects from within Prolog. But in both approaches the programmer has no control over how Java objects are reified on the Prolog side.

Other solutions, based on linguistic symbiosis, have attempted to free programmers from the error-prone task of writing explicit boilerplate code for linking the two worlds. A representative example is the *SOUL* language-(Wuyts 2001), which provides a tight symbiotic integration of a Prolog-like logic language within its host object-oriented language (Smalltalk (Goldberg and Robson 1983)). In SOUL, the unification of arbitrary objects can be customised by overriding certain methods in the classes of such objects (Brichau et al. 2007). Although conceptually different, customising the logic unification of an object has, in certain practical senses, similar effects as customising its reification as a standard Prolog term. A significative advantage of the SOUL approach is, however, that an explicit causal connection between a Smalltalk object and its term representation is maintained. This facilitates the further processing of query results, since real Smalltalk objects (not just their term representation) are bound to logic variables. However, a limitation of this technique is that an arbitrary object has a unique unification strategy (i.e., it is not context-dependent), instead of allowing its unification to vary according to the requirements of a specific problem. Although Smalltalk could allow for the dynamic (re)definition of methods (in this case the methods defining the unification strategy), SOUL currently does not provide a simple high-level mechanism to change the logic reification of an object according to its particular context of use.

In addition to SOUL, other techniques exist that use advanced linguistic symbiosis for analysing object-oriented programs (e.g., (De Volder 2006; Semmle Ltd. 2010)). However, the focus of these techniques is on querying (or transforming) object-oriented programming artefacts from the logic side and do not provide an automatic and transparent linguistic symbiosis from the object-oriented language perspective.

There are a number of other works attempting to provide a symbiotic integration between object-oriented languages and Prolog. Most of them do this from the perspective of the logic language, mainly by offering a set of built-in Prolog predicates that enables easy access to the object-oriented language. Examples include Java, Scala, and Python implementations of Prolog (Chirico 2012; Boulanger and Geske 1998; Friedrich Bolz 2009; Paul Tarau 2012). In the best cases, libraries for communication from the object-oriented language back to the logic world are provided, but they fail to abstract the programmer from low level mappings, requiring an explicit representation of logic concepts (logic engine, queries, logic terms) in the object-oriented program.The same problem occurs for rule engines embedded in Java, like the one proposed by (Friedman-Hill 2003), that use a declarative language other than Prolog.

An interesting mapping technique from methods to logic predicates using method type parameters and annotations is presented in (Cimadamore and Viroli 2008). The main shortcoming of this approach is that the types participating in the declaration of symbiotic methods have to be logic term types. Therefore, there is no implicit mapping between objects and their term representations, but term objects must be explicitly created every time a method is invoked.

Another interesting approach that integrates Java with a logic constraint solver is presented in (Majchrzak and Kuchen 2011). This work relies on a symbolic virtual machine and the syntax of Java programs is left unmodified. Methods evaluated as logic computations are identified with an annotation. Logic variables are also identified with annotations and are limited to Java primitive types. A limitation is the lack of adaptation of the result of a logic method as in our approach; instead all logic methods must return an object instance of class `Solutions`.

Into another category, there are approaches that integrate the two paradigms extending the syntax of the object-oriented language to include constructs and concepts of the logic language. Some of these techniques are focussed in allowing the embedding of Prolog code into Java programs, but they do not provide any automatic mapping between Java objects and logic terms (e.g., Prolog lists are declared with a special syntax and are not the same than Java lists) (Miklós Espák 2006). We think these approaches converge more to the definition of a new hybrid language supporting the two paradigms than to the implementation of transparent and automatic linguistic symbiosis.

3 Research Goal

We envision a conceptual framework and its corresponding implementation that simplifies the implementation of hybrid systems composed of both logic and object-oriented modules. We intend to reduce, or completely eliminate, the amount of interoperability boilerplate code that currently needs to be written, both from the object-oriented and logic perspective. Instead, this interconnection code should be generated by means of a highly customisable inference process based on reflection, reducing the programming effort in such an error-prone and repetitive activity.

In addition to simplifying how routines from the foreign language are invoked, we intend to provide mechanisms to give a programmer a fine-grained control over how Java objects are reified in Prolog, and how Prolog terms are expressed as objects in Java.

Finally, we intend to make our work portable enough so our framework can be used with different Prolog engines, and thus serve as the basis for the construction of common reusable hybrid components that may be useful to the entire Prolog community.

4 Research Status

This research is actively being developed as the central topic of my PhD thesis. I have made a study of related work that, although broad, still needs to be expanded. I have developed initial prototypes of my framework that have been discussed in international venues (Castro et al. 2013; Castro et al. 2012). The case studies were based on a Java program querying a knowledge base modelling an underground system (example adapted from (Flach 1994)). We are currently writing more sophisticated examples that can help us to illustrate the advantage of linguistic symbiosis. One of them is an hybrid Java-HTML-Prolog application that allows a user to query geographical data using Prolog, and visualise the results of his queries in a map (by means of Google Maps). Another example is a portable Prolog query browser with a Java interface, allowing to select different Prolog engines and make queries over them using a unified user-interface. In addition to these case studies, additional work built on the lessons learned in our first experiments is currently submitted to other venues and under writing process.

5 Preliminary Results

In (Castro et al. 2013; Castro et al. 2012) we presented an instantiation of our technique as a framework called *LogicObjects*, which uses linguistic symbiosis to provide a two-way mapping between the Java world and the logic world. Our focus was on attaining a transparent and semi-automatic integration of Prolog within the Java language by providing a simple mechanism for inferring the appropriate mappings between Java and Prolog artefacts. Default mappings were defined for typical scenarios and a mechanism of customising these mappings, by means of annotations, was provided.

Although we accomplished our objectives, our framework was difficult to extend to other customisation mechanisms (different from the built-in one based on annotations) and not portable to Prolog engines other than the ones we targeted. In addition, our implementation was rather hard to maintain. This was mainly because methods dealing with artefact conversions (i.e., the mechanism defining how an artefact in one language should be represented into the other) had to analyse a considerable amount of conditions to infer the appropriate conversion to apply. This caused a high cyclomatic complexity in the implementation, aggravated by the fact that the right conversion, as we demonstrated, often depends on a specific usage context. Furthermore, conversion algorithms were often tangled with other unrelated concerns. Building upon the difficulties we faced and the lessons we learned, we have implemented a library (currently under submission) for modularising context-dependent conversion concerns between inter-language artefacts.

We also have decoupled our approach from a specific bridge library. Instead, it interacts with an abstraction of a Prolog engine that communicates with a vendor specific engine by means of lightweight drivers. In this way, we have designed a portable layer for interacting with distinct Prolog engines, a bit in the style of JDBC. We have currently implemented drivers for JPL (Singleton et al. 2004), InterProlog (Calejo 2004) and the PDT Connector (Rho et al. 2004) bridge libraries. By means of these drivers, currently we are able to interact with SWI (Wielemaker et al. 2012), YAP (Costa et al. 2012) and XSB (Swift and Warren 2012) Prolog.

Finally, we have started the development, on top of our portable layer, of reusable hybrid components (specially user-interface components). We hope this work may be helpful to members of the community willing to speed up the process of creating hybrid Prolog-Java applications.

6 Open Issues and Expected Achievements

Most of our current work has been based on enabling transparent and automatic linguistic symbiosis support from the perspective of a Java programmer. To implement equivalent support from the Prolog perspective, we plan to develop a portable library on top of Logtalk (Moura 2003), equivalent to the existing on the Java side. We believe Logtalk will provide us with the reflective and portability features that we require for this task.

Many of the technical difficulties we face concern about implementing a common API on top of existing bridge libraries. This is because these libraries have different capabilities, advantages and shortcomings. With certain probability, not all the functionality in our common API is going to be available for all our drivers, but we intend to look for techniques that can minimise this problem.

Regarding the validation of our framework, it is not an easy task to find a single case study that illustrates all the advantages of our approach. Instead, we have opted to illustrate it through a variety of small to medium-sized examples, as discussed in section 4. If timing allows us, we would also like to conduct a larger experiment with systems that may benefit from inter-language symbiotic reflection (Gybels et al. 2006), in particular a system using one language to query and modify artefacts in the other language using linguistic symbiosis. For instance, we may consider applying linguistic symbiosis to existing techniques for querying and modifying Java source code artefacts through Prolog (Kniesel et al. 2007; Roover et al. 2011; Mens et al. 2001).

A non-technical difficulty concerns licensing issues, given the nature of the framework we are implementing. We intend to release our work as open source to the community (the exact license still remains to be determined). However, the drivers developed on top of our library may be considered derivative work of existing bridge libraries. If the license of these libraries has binary distribution constraints, our drivers cannot be packaged together with such libraries and the installation process will be more complex, specially because there can be mismatches between the version of the bridge library used to develop the driver and the latest available version of such bridge library.

Regarding our academic contribution, we still have many open questions concerning the best way to conceptualise our approach so it can be extrapolated to other problems that may benefit from linguistic symbiosis techniques. There is still pending to develop a clear, exhaustive, and structured analysis of the possible problems that can be encountered when mapping between an object-oriented and a logic programming language, and of the possible mappings between inter-language artefacts. It also remains pending an implementation independent discussion on how to represent these mappings at a conceptual level, together with an exhaustive study of possible mapping techniques.

References

Boulanger, D. and Geske, U. 1998. Using Logic Programming in Java Environment (Extended Abstract). Tech. Rep. 10, Knowledge-Based Systems Group, Vienna University of Technology, Austria.

Brichau, J., De Roover, C., and Mens, K. 2007. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*.

Calejo, M. 2004. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. In *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, José Júlio Alferes and João Alexandre Leite, Ed. Lecture Notes in Computer Science, vol. 3229. Springer, 714–717.

Castro, S., Mens, K., and Moura, P. 2012. LogicObjects: A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'12)*.

Castro, S., Mens, K., and Moura, P. 2013. LogicObjects: Enabling Logic Programming in Java Through Linguistic Symbiosis. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL)*. Rome, Italy.

Chirico, U. 2012. Jiprolog. http://www.jiprolog.com/documentation.aspx.

Cimadamore, M. and Viroli, M. 2008. Integrating Java and Prolog Through Generic Methods and Type Inference. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*. ACM, 198–205.

Costa, V. S., Rocha, R., and Damas, L. 2012. The YAP Prolog System. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

De Volder, K. 2006. JQuery: A generic code browser with a declarative configuration language. In *PADL*. 88–102.

D'Hondt, Maja and Gybels, Kris and Jonckers, Viviane. 2004. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM symposium on Applied computing*. SAC '04. ACM, New York, NY, USA, 1328–1335.

Flach, P. 1994. *Simply Logical: Intelligent Reasoning by Example*. John Wiley & Sons, Inc., New York, NY, USA.

Friedman-Hill, E. 2003. *Jess in Action: Java Rule-based Systems*. Manning, Greenwich, CT.

Friedrich Bolz, C. 2009. Pyrolog: A Prolog interpreter written in Python using the PyPy translator toolchain. https://bitbucket.org/cfbolz/pyrolog/.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gybels, K. 2003. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*.

Gybels, K., Wuyts, R., Ducasse, S., and D'Hondt, M. 2006. Inter-language reflection: A conceptual model and its implementation. *Comput. Lang. Syst. Struct. 32,* 2-3 (July), 109–124.

Ichisugi, Y., Matsuoka, S., and Yonezawa, A. 1992. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel. In *International Workshop on New Models for Software Architecture (IMSA): Reflection And Meta-Level Architecture*. 24–35.

Kniesel, G., Hannemann, J., and Rho, T. 2007. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*. LATE '07. ACM, New York, NY, USA.

Majchrzak, T. A. and Kuchen, H. 2011. Logic java: combining object-oriented and logic programming. In *Proceedings of the 20th international conference on Functional and constraint logic programming*. WFLP'11. Springer-Verlag, 122–137.

Mens, K., Michiels, I., and Wuyts, R. 2001. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*.

Meyer, B. 1988. *Object-Oriented Software Construction*, 1st ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Miklós Espák. 2006. Japlo: Rule-based programming on java. *Journal of Universal Computer Science 12,* 9 (sep), 1177–1189.

Moura, P. 2003. Logtalk – Design of an Object-Oriented Logic Programming Language. Ph.D. thesis, Department of Computer Science, University of Beira Interior, Portugal.

Paul Tarau, P. 2012. Styla: a lightweight Scala-based Prolog interpreter based on a pure object oriented term hierarchy. http://code.google.com/p/styla/.

Rho, T., Degener, L., Günter Kniesel, Frank Mühlschlegel, Eva Stöwe, Noth, F., Becker, A., and Alyiev, I. 2004. The Prolog Development Tool - A Prolog IDE for Eclipse. http://sewiki.iai.uni-bonn.de/research/pdt/.

Roover, C. D., Noguera, C., Kellens, A., and Jonckers, V. 2011. The soul tool suite for querying programs in symbiosis with eclipse. In *PPPJ*. 71–80.

Russel, S. and Norvig, P. 1995. *Artificial Intelligence, A Modern Approach*. Prentice Hall.

Semmle Ltd. 2010. SemmleCode. http://semmle.com/.

Singleton, P., Dushin, F., and Wielemaker, J. 2004. JPL 3.0: A bidirectional interface between Prolog and Java. http://www.swi-prolog.org/packages/jpl/java_api/.

Swift, T. and Warren, D. 2012. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming 12,* 1-2, 157–187.

Wielemaker, J. and Costa, V. S. 2011. On the portability of prolog applications. In *PADL*. 69–83.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

Wuyts, R. 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Vrije Universiteit Brussel.

Wuyts, R. and Ducasse, S. 2001. Symbiotic Reflection between an Object-Oriented and a Logic Programming Language. *International Workshop on MultiParadigm Programming with Object-Oriented Languages*.