Reasoning Module Interface Design Considerations

Alejandro G. Stankevicius*

Laboratorio de Investigación y Desarrollo en Inteligencia Artificial
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca - Buenos Aires - ARGENTINA
e-mail: ags@cs.uns.edu.ar

Abstract

DLP is a basic java-based IDE specifically conceived for DeLP, that allows to compile DeLP programs and queries into JAM opcodes, execute those opcodes, and even witness the actual step by step construction of the dialectical tree that is built in order to determine the correct answer to those queries.

Over time, DeLP semantics matured into a solid alternative for representing the knowledge of an autonomous agent and also for deriving new conclusions from this knowledge. In this article we address how can this particular DeLP implementation be wrapped or encapsulated in a way that simplifies the construction of this kind of agents, that is, agents that use DeLP as their Knowledge Representation and Reasoning (KR&R) framework.

1 Motivations

DLP is a basic JAVA-based IDE (Integrated Development Environment) specifically conceived for DeLP [2] (Defeasible Logic Programming), that allows to compile DeLP programs and queries into JAM opcodes,[1] execute those opcodes, and even witness the actual step by step construction of the dialectical tree that is built in order to determine the correct answer to those queries.

This project was initially aimed at honing our own understanding of the interaction among the arguments considered throughout the dialectical analysis required to answer queries in a given DeLP program. An overview of its architecture can be found in [6], and some the techniques devised for compiling this kind of code were reported in [7]. We found that having this visualization aid also allowed us to evaluate tentative changes to the formalism semantics just by observing their impact on the construction process of the dialectical trees.

Over time, DeLP semantics matured into its latest manifestation [3], a change followed by a matching update to the abstract machine implemented within this IDE, briefly stated in [5]. At this point, DeLP became an solid alternative for representing the knowledge of an autonomous agent, and also for deriving new conclusions from this knowledge (a possibility outlined in [4]). In this article we address how can this particular DeLP implementation be wrapped or encapsulated in a way that simplifies the construction of this kind of agents, that is, agents that use DeLP as their Knowledge Representation and Reasoning (KR&R) framework. In order to do so, we first overview of the

*Partially supported by CIC (*Comisión de Investigaciones Científicas de la Provincia de Buenos Aires*).

[1] a WAM like virtual machine suitable for DeLP.

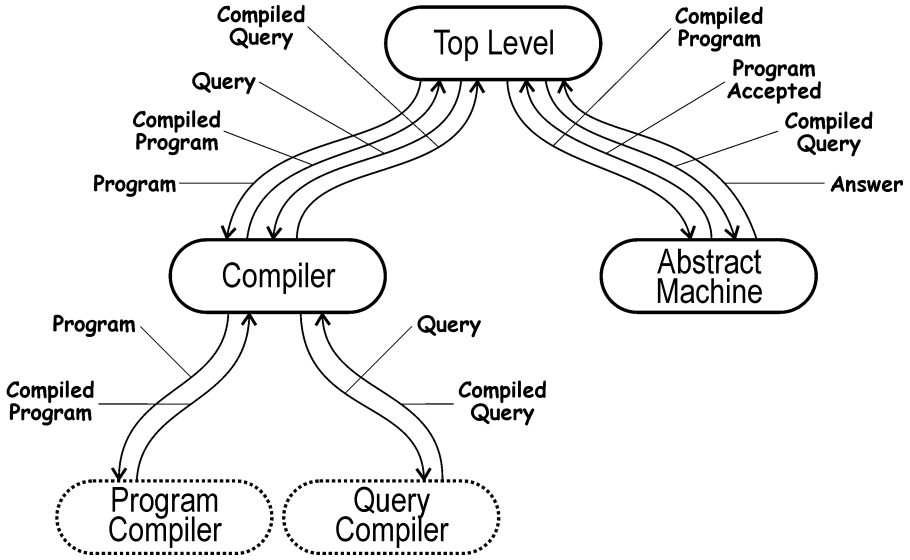

Figure 1: DLP architecture

architecture of DLP and the role of its main components, and then discuss the design considerations that a reasoning module based upon it should observe.

2 DLP Architecture

Figure 1 depicts an overview of the architecture that DLP implements. It encompasses three main components: the *compiler*, the *abstract machine*, and the *top-level* governing the interaction between them. This top-level is in charge of controlling the flow of information within the system (in the figure above, the labels of the arcs connecting these components). Simply put, the top-level feeds the abstract machine with the compiler output, both for programs and queries. Another task delegated on the top-level is the interaction with the user, both when accepting DeLP programs and queries, or when displaying back the corresponding answers. The compiler, in turn, covers two main tasks: compiling DeLP programs and compiling queries. Even though these tasks share a lot in common, some key characteristics differ (for instance, DeLP programs require generating consistency checking code, while queries do not). Finally, the abstract machine actually executes the string of opcodes obtained as a result of the compilation of programs and queries. When receiving a compiled program, it merely loads it into its memory, but when receiving a compiled query, it has to return not only the actual answer, but also additional information regarding the actual argument found by the abstract machine supporting the answer just provided. The architecture of this abstract machine, first introduced in [1], is quite straightforward: a group or registers interact with a set of memory regions by virtue of a set of instructions (called opcodes). These memory regions allow the representation of different elements of DeLP programs, such as program code, the execution stack, the current line of argumentation, etc.

3 Design Considerations for the Reasoning Module Interface

Designing the interface of a *generic* reasoning module is a daunting task. It is quite difficult to circumscribe the effect of the decisions about how knowledge should be represented to a single module, given that *what an agent can do* is tightly intertwined with *the knowledge it posses* about

the world around it. For instance, whether an agent can plan its actions ahead of time might depend on the complexity of its own knowledge representation. In this work we deal with a simplified scenario, as we already know which language shall be used. In fact, we even have a tentative implementation of the reasoning module (briefly stated in the previous section), so all we need to do is weigh what would a reasonable interface for it be, considering that instead of a users now we have other modules of an hypothetical agent.

To begin with, this problem can be tackled from two incompatible positions:

- Keep the interface as minimal as possible.
- Take advantage of the wealth of information generated by the abstract machine (for instance, all the steps in the construction of the dialectical tree of a given query).

Minimizing the size of a module interface is a well known and sound principle, but the additional information generated by the abstract machine is also present there, and it is available at no extra cost. There must be specific instances where this additional information might be of use, but in general terms, what really matters in a reasoning module is essentially to provide answers to queries. This observation favors the first approach, so out interface must solely provide a mean of accepting DeLP programs and queries, and returning answers.

Both DeLP programs and queries are just plain text ($i.e.$, a string), but answers to queries containing variables should also make available the substitutions found while solving them. Moreover, the same query might also have several different substitution (just like in Prolog, where the same query might be solved using multiple substitutions). To sum up, the interface must provide the following services:

1. `loadProgram(String P) : Boolean;`
2. `solve(String Q) : Boolean;`
3. `nextAnswer() : Answer;`
4. `hasNextAnswer() : Boolean;`
5. `getCurrentBindings() : Bindings;`

The boolean value returned by `loadProgram` and `solve` denote whether the program or query provided was successfully parsed or an error was found. Even though the compiler provides additional information regarding where the syntax error occurred, since we are pursuing a minimal interface a bare boolean value should do. The third service, `nextAnswer`, allows the inspection of the answer to the last query posed. It returns an `Answer`, which can be compared against a set of predefined values (such as `Answer.YES`, `Answer.NO`, etc), or converted into a string using the standard `toString()` method. This service (`nextAnswer`) can be requested only when `hasNextAnswer` already confirmed the existence of another answer (for instance, right next after calling `solve`). Finally, `getCurrentBindings` provides the bindings associated with the most recent answer to the last query, as provided by `nextAnswer`. This service structures all the bindings corresponding to this answer in an abstract list, an object implementing the `Iterator` interface. This interface allows an easy traversal of each individual binding, which might later be converted into a string using, once again, the `toString()` method.

4 Summary

In this article we briefly overview the architecture of an implementation of DeLP, and began considering how it can be wrapped or encapsulated in a way that simplifies the construction of agents that use DeLP as their Knowledge Representation and Reasoning (KR&R) framework. In order to do so, we identified two ways of achieving this goal: either minimizing the size of the interface, or conversely providing as much information as possible. Following the principle of keeping module interfaces small, we proposed five basic services covering all the major aspects of the interaction with a reasoning module built upon these premises.

Finally, it should be pointed out that the latest DLP implementation can always be found at `http://cs.uns.edu.ar/~ags/DLP`. It is currently free software (that is, free there is used as in 'free beer'), but we are considering releasing it as software *really* free (in this case, we mean free as in 'free speech'), under the well known *GNU General Public License*, once we tidy up its source code.

References

[1] García, A. J. La Programación en Lógica Rebatible: su definición teórica y computacional. Master's thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, June 1997.

[2] García, A. J. *Programación en Lógica Rebatible: Lenguaje, Semántica Operacional, y Paralelismo.* PhD thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, Dec. 2000.

[3] García, A. J., and Simari, G. R. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming 4*, 1 (2004), 95–138.

[4] Stankevicius, A. G., Capobianco, M., and Chesñevar, C. I. An architecture for rational agents interacting with complex environments. In *Proceedings of the V Workshop de Investigadores en Ciencias de la Computación (WICC)* (Tandil, May 2003), Universidad Nacional del Centro de Provincia de Buenos Aires, pp. 551–555.

[5] Stankevicius, A. G., and García, A. J. An Abstract Machine for the Execution of DeLP Programs. In *Proceedings del 10mo Congreso Argentino de Ciencias de la Computación (CACiC)* (La Matanza, Oct. 2004), Universidad Nacional de la Matanza, pp. 1530–1541.

[6] Stankevicius, A. G., García, A. J., and Simari, G. R. Una arquitectura para la ejecución de Programas Lógicos Rebatibles. In *Proceedings of the 5th International Congress on Informatics Engineering* (Capital Federal, Argentina, Aug. 1999), Universidad de Buenos Aires, pp. 450–461.

[7] Stankevicius, A. G., García, A. J., and Simari, G. R. Compilation Techniques for Defeasible Logic Programs. In *Proceedings of the 6th International Congress on Informatics Engineering* (Capital Federal, Argentina, Apr. 2000), Universidad de Buenos Aires.