INFORMATICA, 2012, Vol. 23, No. 2, 203–224

Incorporating the Ontology Paradigm into a Mainstream Programming Environment

Dragan DJURIC, Vladan DEVEDZIC
FON, University of Belgrade
Jove Ilica 154, 11000 Belgrade, Serbia
e-mail: dragan@dragandjuric.com

Received: August 2010; accepted: November 2011

Abstract. The emergence of the Semantic Web have revived the interest in knowledge engineering and ontologies. Different paradigms often share challenges and solutions, and can complement and mutually improve each other. This paper presents a simple and agile integration of ontologies and programming on a small scale, and in a down-to-Earth manner by incorporating the ontology paradigm into a mainstream programming environment. The approach is based on metaprogramming, which has been used to internalize the ontology modeling paradigm into the Clojure language. The resulting DSL, Magic Potion, is implemented in Cojure and blends ontology, functional, object-oriented and concurrent paradigms, which is suitable for general-purpose domain modeling, from technology enhanced learning to business.

Keywords: programming paradigms, multiparadigm languages, ontology paradigm, ontology languages, metaprogramming, domain-specific languages, programming languages, domain engineering, programming techniques.

1. Introduction

One of the central activities in software development is modeling and implementing business domains. Business domain modeling is essentially a kind of domain knowledge modeling, so knowledge engineering and ontologies can be a solid and sound approach. However, although knowledge engineering has been indirectly influencing software modeling practices for a long time, ontologies and semantic technologies are still rarely used in software development. Current ontology languages and tools neither fit current software development practices well, nor they are easy to use with mainstream programming languages and tools, although different paradigms often share challenges and solutions (Vaira and Čaplinskas, 2011). Moreover, good practices for managing the development of ontologies remain largely vague and are still a research topic (Lavbicand Krisper, 2010).

On the other hand, general-purpose programming languages still do not provide internal support for semantically rich domain-driven programming. Of course, it is possible to extend the grammar of a specific language to include such support. However, extending grammars of widely used languages may not be easy and the relevant community may

not adopt it easily. An alternative can be to develop and use special-purpose libraries (e.g., McBride, 2002), but it always adds complexity and impedance mismatch. Yet another alternative is to use homogeneous metaprogramming. It is a craft and a process of using tools and languages for creating, modifying, adapting, adjusting, and otherwise transforming other programs. Some programming languages support metaprogramming on mainstream platforms in such a way that developers can extend the language/platform with the features they need and implement first-class support for ontology paradigm – to incorporate the ontology paradigm in the host platform.

This paper presents a lightweight metaprogramming approach to bringing semantics and ontologies closer to software engineering environments – incorporating the ontology paradigm into a general-purpose programming language as an internal meta Domain-Specific Language (DSL; Djuric and Devedzic, 2010).

The approach is based on the application of metaprogramming to incorporate the ontology modeling paradigm into a programming environment based on Java ecosystem, as an embedded domain-specific language for modeling business domains. It relies on the use of Clojure, an emerging language for Java Virtual Machine (JVM) that offers homogeneous metaprogramming support (Hickey, 2008). This work is conceptually guided by *Modeling Spaces*, an abstract framework for studying heterogeneous modeling problems in a more uniform way (Djuric *et al.*, 2006).

The objective was to find a way to introduce support for semantically rich domain-driven programming in the host environment in a pragmatic, down-to-Earth manner, suitable for small development teams with limited resources. The result is an internal meta domain-specific language, called Magic Potion, that blends ontology, functional, object-oriented and concurrent paradigms to offer a concise means for developing business domain models.

After the introduction, the second section gives a brief overview of the foundations and enabling technologies that this work is based on. The third section identifies some typical areas in software engineering that ontologies can improve, and define requirements that a solution should fulfill. The fourth section presents *Magic Potion*, a meta DSL that incorporates the ontology paradigm in Clojure language and Java platform to support semantically rich means for business domain modeling/programming. The fifth section discusses the application of the presented approach in some software engineering tasks beyond domain modeling. The sixth section presents the results of an evaluationof the approach, and the seventh brings the discussion on its suitability for software engineering. The paper uses an example of an art dealership business domain that is simplified but still tailored to illustrate the elegance and practicality of the presented solution for difficult modeling tasks.

2. The Foundation and Related Work

2.1. *Modeling Spaces*

A domain model is a conceptual model of a system. It abstractly represents the system, usually by describing entities and their relations (Gasevic *et al.*, 2009). Mainstream soft-

ware development methodologies (Larman, 2004) often incorporate domain layer at the core of their architecture as the representation of the real-world concepts (e.g., business-related), and build the supporting infrastructure around it. A typical infrastructure includes the orthogonal aspects of security, persistence, communication, distributed and parallel computation, etc. These aspects are also based on their own orthogonal models of the respective domains.

Models, in their broadest meaning of being abstract representations of real-world things, are built using modeling languages. Here, any computer program is considered a model of the real world and thus any programming language a modeling language. Many heterogeneous languages often interoperate and are used at many levels of abstraction or to define one another. The description of the approach has been based on Modeling Spaces (Djuric *et al.*, 2006), an encompassing framework for studying heterogeneous modeling and meta-modeling problems inspired by Model-Driven Architecture (MDA; Schmidt, 2006).

Model-Driven Architecture (Schmidt, 2006) is an ongoing software engineering effort driven by Object Management Group (OMG). It defines three viewpoints (levels of abstraction) from which a certain system can be analyzed. Starting from a specific viewpoint, the following system representations (viewpoint models) can be defined: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). MDA is based on a four-layer meta-modeling architecture that has been used as an inspiration and was further generalized to n-layered modeling architecture by Modeling Spaces framework.

Figure 1 shows a general n-layered modeling architecture (Gasevic *et al.*, 2009). The M0 layer is the real world, abstractly represented using models (M1 layer). Models are created using concepts defined in metamodels (M2), which are created using concepts defined in meta-metamodels (M3). The topmost layer contains the super-metamodel (Mn), which is metacircular (defined by its own concepts).

The term *represents* denotes that models stand in place of real-world things, acting on their behalf in some specific context. Models' concepts *conform to* metaconcepts that define them, in the sense that metaconcepts determine their nature, specify their precise meaning and form, and identify essential qualities. Each paradigm has its own variations of meaning for these terms.

A *Modeling Space* (MS; Djuric *et al.*, 2006) is a modeling architecture based on a particular super-metamodel. Every layer above M0 in this hierarchy conforms to the higher layer, finally reaching the top layer containing the self-defined super-metamodel.

The important thing to understand is that this architecture depends on the context; the position of some thing is not absolute. In a multiparadigm approach, there are many modeling spaces. Some models represent the same thing independently, i.e., from different perspectives (*parallel spaces*), or a model from one modeling space can represent something from another space (*orthogonal spaces*).

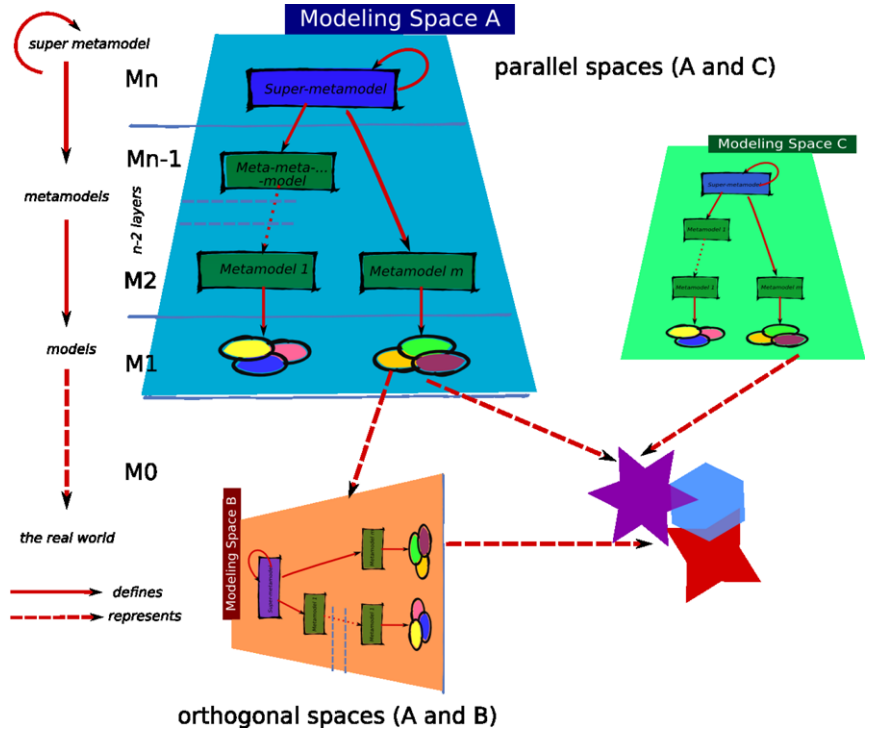

Fig. 1. Multi-layer modeling architecture.

2.2. *Metaprogramming*

Metaprogramming is a process of making programs (*metaprograms*) that manipulate other programs as data (*object programs*) (Sheard, 2001). Typical metaprogramming includes:

- dynamic generation of source code (SQL expressions, scripts);
- using built-in language extension mechanisms (Java annotations);
- metaprogramming languages that extend the host language or create a new one;
- creating compilers, etc.

Metaprogramming languages can be *heterogeneous*, when the meta-language is different from the object language and *homogeneous*, when the meta-language and the object language are the same. Heterogeneous languages (TXL, Stratego/XT, ML etc.; Sheard, 2001) usually offer more possibilities than homogeneous ones, but require very specialized knowledge of both the metaprogram and the object program internals and are more difficult to learn and use.

Recently, general purpose languages with metaprogramming capabilities have started to attract a substantial attention. They offer more metaprogramming support than mainstream languages like Java or C#. Clojure, Ruby, Python, Perl, Lua and other dynamic languages offer various metaprogramming techniques.

2.3. *Domain-Specific Languages*

A Domain Specific Language (DSL; Van-Deursen and Visser, 2000) is a computer language targeted at a specific kind of problem, rather than any kind of problem as is the case with general-purpose computer languages. DSLs are, by design, simpler than general-purpose languages. DSLs have been around for quite some time, but have recently become more popular due to the rise of domain-specific modeling and model-driven engineering. The concept of a DSL is quite close to that of ontology – both provide terminology for representing specific problem domains.

DSLs can be *internal* (embedded, homogeneous), when the facilities of the host language are utilized to create a DSL that also conforms to the host language, and *external* (heterogeneous), which use their own custom syntax and require building a special parser and tools (Langlois *et al.*, 2007). Typical examples of internal DSLs are expectations in JMock, parts of Ruby on Rails framework, and many libraries in Lisp; external DSL examples are CSS, SQL, and ant.

2.4. *Clojure*

Clojure (Hickey, 2008) is a recently developed language that compiles directly to the Java or CLR bytecode. Being a dialect of Lisp language, it inherits its simplicity, high expressiveness and adaptability, as well as a strong theoretical foundation. Better yet, it is a pragmatic language that leaves out historical intricacies of Lisp while embracing modern mainstream platforms (Java and .NET), allowing seamless integration with their ubiquitous libraries.

Clojure is a functional language that models the real world as a set of functions that take certain values and produce others. Pure functions have no side effects, they do not change any external memory. In contrast to today's prevailing imperative style, they do not have any effects on memory. To the extent the program is fully functional, it does not modify anything that is not local to the function, and thus there is no need for synchronization.

As in other powerful functional languages, Clojure code can define functions that create other functions, functions that receive other functions as parameters, and functions that combine other functions with a set of variable bindings, called closures. Clojure also fully supports macros. Macros are essentially functions that generate Lisp (Clojure) code; they are programs that write other programs. These powerful facilities that are awkwardly implemented and rarely found in mainstream programming languages are essential for writing embedded DSLs (VanDeursen and Visser, 2000).

Transactional memory is similar to concurrency control found in database transactions, where concurrent access to a shared memory is controled by transactions that guarantee atomicity. Clojure has a built-in support for concurrency via an implementation of *software transaction memory* (Shavit and Touitou, 1997) that protects *mutable references* that can change the *immutable values* (represented by various data structures) they hold only inside a transaction.

More details on Clojure and Lisp can be learned from Halloway (2009), Graham (1993), Hickey (2011) and Volkmann (2009).

2.5. *Description Logics and Ontologies*

Description Logics (DL) languages are formalisms for representing knowledge (Baader *et al.*, 2007). This is one of the main theoretical cornerstones of ontologies and the Semantic Web (Berners *et al.*, 2001), the intelligent layer of the largest distributed information system in the world – the World Wide Web. Ontological languages are very descriptive and theoretically sound modeling languages. The Semantic Web standard ontology language is Web Ontology Language (OWL; Motik *et al.*, 2008), an extension of RDF (Klyne *et al.*, 2004). However, the Semantic Web technologies are still too resource-demanding and developer-unfriendly to reach the envisioned usability. Still, they have popularized quite a few good ideas that this work aims to make more practical and approachable for software engineers through this research.

The main constructs in DL theory are atomic concepts, as unary predicates, and atomic roles as binary predicates on the given domain. Complex concepts and roles are defined over atomic ones using logical constructors like negation, intersection, union, etc. For example, if *Customer, Artist* are atomic concepts on the domain of all people in the world (Customer $\sqcap \neg$ Artist), is a complex concept of all customers that are not artists. Such languages are equipped with a formal logic-based semantics.

A great advantage of DLs is availability of reasoning mechanisms usually based on *tableau* or *hyper-tableau* (HT) algorithm (Motik *et al.*, 2007). Tableau is a graph and is designed for concept satisfiability. A HT algorithm is applicable to DLs knowledge bases (KB) extended with description graphs (DGs) . It is also a classification algorithm.

3. Ontology-Based Software

3.1. *Domain Modeling and Ontologies*

Currently, the most popular mainstream platforms are centered around the object-oriented (OO) paradigm. Object-oriented programs are supposed to be based around well defined and flexible objects that expose flexible behavior while hiding the limiting internal implementation and actual data. However, OO approach has lately been heavily criticized for its poor suitability for building parallel applications; the practical applications often divert to essentially non-OO solutions implemented in OO languages[1] (Fowler, 2004).

In addition, OO suffers from other serious limitations (Gasevic *et al.*, 2009):

- it models behavior, not the semantics of data;
- it often requires large ceremony and self-discipline for supporting domain-modeling tasks that are not built-in (constraints, complex associations between objects, etc.);
- it is not based on any mathematically sound theory, which leads to ad-hoc implementations that are difficult to analyze;

[1] Anemic Domain Model (anti)pattern coupled with Service Layer pattern are the most obvious example.

- its concept of inheritance, as implemented in mainstream OO languages, is not in accordance with the set theory;
- its mutable objects are not suitable for parallel computations.

On the other hand, business domain modeling is essentially a kind of knowledge modeling (Gasevic *et al.*, 2009), a field of artificial intelligence that became attractive through the Semantic Web movement (Shadbolt *et al.*, 2006). Semantic Web is based on ontologies, a very descriptive, flexible and theoretically sound means for knowledge modeling based on description logics, mathematical formalisms for representing knowledge. However, XML-based Web Ontology Language (OWL; Shadbolt *et al.*, 2006) ontologies have some practical drawbacks. They are not executable, which means they are stored in repositories written in, say, Java and have to be accessed and manipulated through repository APIs (Jena, jena.sf.net), which makes them infrastructure-demanding and programmer-unfriendly.

The objective is to take what's best from the ontology paradigm (the expressiveness and the theoretical background of description logics) and implement it natively in Clojure to support business domain-driven programming, making it accessible to all JVM-based programs. The implementation has to blend the ontology paradigm into the existing environment in a programmer-friendly manner that would feel like the new paradigm is natively supported in Clojure.

3.2. *Enhancing Software with Ontologies*

A programming language that natively supports ontologies as a natural means for domain-driven programming in a programmer-friendly manner can improve software engineering practices in the following ways:

- *Semantics-based software development*: A programming language that natively supports semantically rich means for business domain modeling can help programmers adopt semantic-based approach to programming (Djuric and Devedzic, 2010; Djuric *et al.*, 2010). Current programming languages do not offer such capabilities. They are sometimes available through library support, which impose a high impedance mismatch: they are not easy to learn and use, and have poor performance.
- *Integration of software and semantic technologies*: Native support for semantics through incorporating the ontology paradigm brings a much closer integration with other paradigms that are built into the language or are incorporated (Djuric and Devedzic, 2010).
- *Connection to the Semantic Web languages*: The incorporated ontology paradigm is much closer to RDF and OWL than Java, C#, Python, Ruby or any other widespread language is (Djuric and Devedzic, 2010).
- *Component discovery and ontologies*: In this area, semantic technologies are primarily used for Semantic Web services (Sycara *et al.*, 2003). The incorporated paradigm, being the new integral part of the host environment, can be non-intrusively used for building a specialized ontology in the form of a DSL for describing the program itself: different modules, components and other parts.

- *Feature modeling and ontologies*. The incorporated ontology meta-DSL can be used to build a special DSL for semantic feature modeling (Peng *et al.*, 2006) in a host language that feels natural to software developers.
- *Ontology reasoning for software engineering*: The models built with the incorporated ontology language can be used as a source for custom-built reasoners, or can be transformed to standard ontology languages and used as an input for standard reasoners (Sirin *et al.*, 2007; Tsarkov and Horrocks, 2006) easier than mainstream non-semantic language constructs.
- *Semantic annotations in software engineering*: Semantic annotation is about assigning to the entities in the text links to their semantic descriptions. The programs written in the incorporated language seamlessly integrate with other parts written in the host language, and can be used to seamlessly annotate other parts of the program.
- *Ontology-driven software architectures*: Most current research in this direction is geared towards semantic web services (Paolucci *et al.*, 2002; Sycara *et al.*, 2003). Business domain layer is one of the cornerstones of the popular multi-layer software architecture (Fowler, 2004). Basing this layer on the ontology paradigm lays the foundation for a more general ontology-driven architecture.Suitability of the Incorporated Paradigm

The following key points need to be addressed in developing this DSL (Djuric and Devedzic, 2010; see also Spinellis' (2008) requirements):

- *Homogeneous approach*. Such a DSL should be implemented using the same platform/language as the native one, thus lowering the increase in complexity incurred by using multiple paradigms.
- *Minimum of additional features*. This DSL should implement only the features of ontologies needed for the most common tasks in the host language. With this approach the disturbance in the application architecture and in the development process itself tend to minimize.
- *Host platform support*. Since homogeneous DSL introduces only a small number of non-intrusive constructs, the resulting implementation of the ontology paradigm is compatible with standard tools of the host platform.
- *Consistency*. If implemented using a suitable language, such a DSL makes programming with the ontology paradigm almost the same as ordinary programming and thus transparent.
- *Clarity*. Relating different platforms, metamodels, and paradigms can be challenging. It is important that incorporation of any new paradigm creates a clear picture of what it is good for and why it is needed in the host environment. This paper uses Modeling Spaces to clarify what's happening during incorporation of the ontology paradigm.
- *Familiarity*. The ontology paradigm is not commonly used in programming, and many developers may not be familiar with it. To ensure easier adoption, the DSL should provide an easy integration of the ontology paradigm with the host environment, in a way that makes the users of the host environment feel comfortable when using it.

- *Tools and services*. Availability of tools and consulting services pose a problem with emerging technologies. The incorporated ontology paradigm and the corresponding DSL should require as simple tools and support as possible. An important point to take into account is whether the tools are open source and free or not.
- *Trends*. The resulting solution should support current trends and developers' interests, and should be supported by available literature and resources.

4. Magic Potion: Ontology Paradigm Incorporated into a General-Purpose Programming Language

4.1. *Magic Potion*

Magic Potion, a meta DSL that introduces the ontology paradigm for business domain modeling into Clojure, is theoretically based on description logics. As Fig. 2 shows, Clojure is used in the Java technical space (Djuric *et al.*, 2006; Gasevic *et al.*, 2009) to build Magic Potion as a DSL for creating other, concrete DSLs related to real-world domains. Magic Potion:

- enables capturing the semantics of business processes (through ontologies);
- seamlessly fits the concepts of rich domain modeling into the concurrent programming paradigm based on Clojure's Software Transaction Memory;
- is practical and easily comprehensible for software developers, requiring minimal theoretical knowledge or even less;
- is formally sound, theoretically based on description logics.

In this case, Clojure was a better choice than Scala or JRuby due to the requirements for native parallel programming support and seamless blend into language's preferred

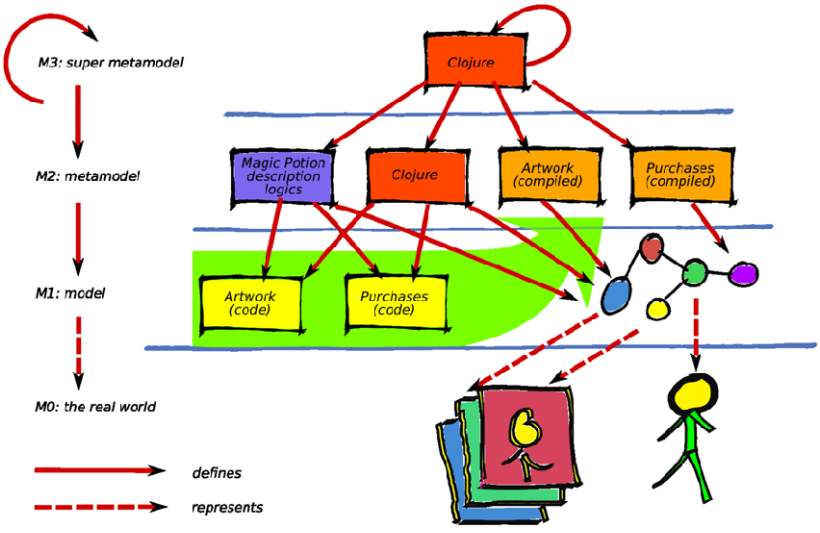

Fig. 2. Magic Potion Architecture.

data structures. It was a better fit for consistency, homogeneous approach, familiarity, and host platform support. Other teams may find that other languages fit their challenges better.

In Clojure's modeling space, Clojure is considered as a super-metamodel at layer M3. Clojure defines all metamodels (programs), even itself. Magic Potion is a Clojure program, a metamodel at layer M2, defined with Clojure functions, macros, and data structures. Using functions and macros defined in Magic Potion, programmers can create models at M1 that describe abstract concepts of some domains and their relations. These models (the code the programmer has written) become really useful only when the programmer compiles them with Clojure compiler. At that moment they become mini-languages that can, when run as programs in memory, describe the customers who have purchased specific paintings under specific policies on specific dates an have paid specific prices. Artwork, Purchases and Customers DSLs become metamodels at layer M2. When executed, they create instances of the respective objects (at M1) that represent real-world artworks, customers and the process of purchasing (M0).

4.2. *Programming with Ontologies*

To illustrate what it means to integrate an ontology paradigm into a host environment, an example from a business domain is used.

Figure 3 shows a UML sketch of a simplified art dealership business domain. Customers purchase items from art dealers. The items could be commodities that can be

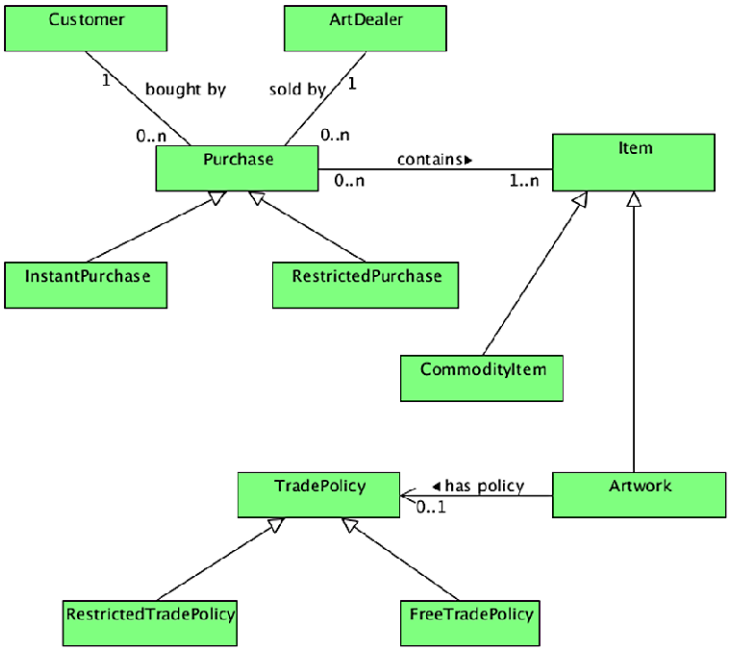

Fig. 3. Art dealership domain.

bought instantly, or artwork that can be under restricted trade. If the artwork is under the restricted policy, the purchase has to get the approval from an authority. This would normally be a part of a much bigger domain that has much more business rules and policies to take into consideration. This favors the declarative programming style, where arbitrary predicates can be freely combined.

Magic Potion enables the programmer to use the description logics abstractions (concepts, properties, roles, restrictions) as if they were parts of Clojure. For example, the following code declares two properties, `aname` and `human-name`, and a concept `customer` that uses `human-name` as one of its roles. Ontology properties are, unlike attributes or associations in object paradigm, independent first-class constructs. Keywords `:restrictions` and `:super` can enhance readability, and are usually not mandatory.

```
(property aname [string?])

(property human-name
   :restrictions [(length-between 2 32)]
   :super [aname])

(concept customer
   [human-name])
```

This domain declaration defines the `customer` function that the programmer can call to create statements about a customer that can have a valid human name, a string between 2 and 32 characters. If she tries to create an invalid statement, she would get a validation exception with a report that contains a list of unsatisfied restrictions for each of the properties. This can be used to produce a nice-looking error report to the user that supplied the invalid data through, say, a Web form.

```
(customer ::human-name "A")

java.lang.IllegalArgumentException
   ([:user/human-name (length_between)])
```

An attempt to create a customer with valid statements would return its representation as a Clojure persistent map.

```
(customer ::human-name "Jason Bourne")
{::human-name "Jason Bourne"}
```

In addition to unqualified restrictions, which apply to a property regardless of the concept in its range, Magic Potion supports qualified restrictions, which apply to a property only in the context of a certain concept. The following declaration of a concept `item` further restricts `aname` only when used as a role of `item`.

```
(concept item
   [(val> aname [(min-length 8) (max-length 256)])])
```

Now the programmer declare properties that can be used to define statements about something that contains `item`, is bought by `customer` and sold by `dealer`, and use them

as roles in `purchase`. Magic Potion supports arbitrary predicates on all statements of a role; e.g., `min-count` constrains multiplicity.

```
(property contains [item?])

(property bought-by [customer?])

(property sold-by [dealer?])

(concept purchase
   [(ref> bought-by)
   (ref> sold-by)
   (ref*> contains [] [(min-count 1)])])
```

Note that, since the domains of these properties are not simple datatypes but individuals represented as immutable maps of statements, the role has been defined to use Clojure refs instead of direct values. To suit parallel computation better, Clojure uses immutable data structures (in this case maps). Once a map is created, it represents a "snapshot" value of an individual that cannot be changed. If direct values have been used, purchase would eternally refer to the specific customer information even when that information becomes outdated (for example, when the customer who moves to a new address makes new purchases).

Refs enable us to create an immutable statement that references another individual that can change its immutable value only in the scope of an STM-managed transaction. `ref>`, `val>`, `ref*>` and `val*>` are functions that create roles that use appropriate referencing and multiplicity one or many.

The next few declarations follow the same principles.

```
(concept trade-policy)

(concept free-trade-policy
   :super [trade-policy])

(concept restricted-trade-policy
   :super [trade-policy])

(property has-trade-policy [trade-policy?])

   (concept artwork
   [has-trade-policy]
   [item])

(concept commodity
   :super [item])

(concept restricted-purchase
   [(ref> approved-by [authority?])]
   [purchase]
```

Clojure multimethods provide polymorphism based on arbitrary functions, not only on parameter type. As the newly implemented ontology paradigm natively blends into Clojure, the programmer can take advantage of multimethods for some of more complex predicates. Depending on the type of the item (in a general case, on any arbitrary function of an item) this tiny artwork DSL enables creating and validating the statements about different kinds of purchases.

```
(defmulti eligible-for-free-trade? type)

(defmethod eligible-for-free-trade?
   :commodity [an-item]
   true)

(defmethod eligible-for-free-trade?
   :artwork [an-item]
   (-> an-item has-trade-policy free-trade-policy?))

(concept instant-purchase
   [(ref*> contains [eligible-for-free-trade?])]
   [purchase]
```

This example demonstrated the use of the ontology paradigm incorporated into a general-purpose programming language. Clojure has a native support for metaprogramming, so Magic Potion actually consists of functions and macros that do not differ at all from regular Clojure code. Since the ontology paradigm sits on top of Clojure's native data structures, the domain functions create validated business objects as statements in ordinary Clojure maps with additional metadata, which are used in a usual Clojure way.

5. Magic Potion Beyond Business Domain Modeling

Although business domain modeling is certainly the central use case for ontologies incorporated into software engineering process, due to their native integration with the host language they can be very useful in other ways as well. In the case of Magic Potion, the following discussion explains these additional use cases:

5.1. *Semantics-Based Software Development*

Magic Potion does not only provide a means to capture domain expert's knowledge in a model that could be transferred to code – it is a way to create *both* a semantically rich domain model and the working code at the same time. It is designed to be friendly to domain experts *and* programmers.

Moreover, it fits well into mainstream agile software development process and supports its practices and tools due to the native integration into Clojure language and Java platform. Magic Potion supports *refactoring*, *testing* with the automatic testing tools, management by *issue tracking integration* tools, automatic *formatting*, formatting compliance, syntax checking, code completion and other techniques and tools available for the host language (Clojure).

5.2. *Integration of Software, Semantic Technologies, and Semantic Web Technologies*

The way that Magic Potion describes the data by the statements is designed to be easily transformed to Semantic Web technologies, notably RDF. For example, the transformation of the following Magic Potion individual (a Clojure map) to RDF is straightforward (RDF is shown in the shorthand Turtle syntax):

```
(def purchase-1 (ref ::bought-by buyer-ref
  ::sold-by seller-ref
  ::contains #artwork-ref-1 artwork-ref-2))

(to-rdf purchase-1)

<purchase-1>
  bought-by "buyer-ref-uuid";
  sold-by "seller-ref-uuid";
  contains ("artwork-ref-1"
    "artwork-ref-2").
```

Concepts and properties can also be transformed to RDF(S) and OWL, but since Magic Potion is based on different kind of DLs than OWL, the transformation at that level is not straightforward and complete. For example, Magic Potion constraint functions have to be translated to OWL restrictions manually, and many of the complex constraints may be not possible to describe in OWL.

5.3. *Component Discovery*

Since Clojure supports metadata, Magic Potion is a natural choice for describing Clojure artifacts with rich semantics. The described components, Clojure functions, protocols, data, agents, references etc. can be later analyzed using that metadata for the purpose of discovering and combining the components. Combined with the transformation to RDF/OWL, such metadata can be read and used by Semantic Web discovery services.

```
(concept policy-processor
  [support-policies authorized-by support-protocols]

(def policy-service (with-meta policy-function
       (policy-processor ...)))
```

Suitability of the component for a particular task can be determined in a straightforward way, through the taxonomy of concepts and properties, or with algorithms that can determine the similarity of interfaces analyzing the domain and the constraint functions.

5.4. *Feature Modeling*

Similarly to the way that general feature modeling is proposed to be supported by Semantic Web technologies (Wang *et al.*, 2005), it can be supported by Magic Potion in Clojure. Standard feature relationships: mandatory, optional, alternative, and additional

constraints: requires and excludes can be supported by an internal DSL created by Magic Potion. The following listing shows a greatly simplified part of such a DSL:

```
(concept feature ...)
(property mandatory [feature?])
(property alternative [feature?])
(concept Concept
   [(ref*> mandatory)
   (ref*> alterantive [] [all-distinct?])])
```

These feature models are used to annotate Clojure artifacts and are suitable for processing by other Clojure and Java code, while still being available to other tools through the transformation to RDF that was previously described.

5.5. *Ontology Reasoning for Software Engineering*

The most straightforward way for the incorporated ontology paradigm to facilitate reasoning on the host platform is through its transformation to RDF/OWL ontologies, which are then used as an input for reasoners that support these technologies. An alternative for Magic Potion is to build a plugin for reasoners such as Pellet, Racer or FaCT that enable them to use Magic Potion data as input. Potentially, the most feature-rich and easy-to-se approach is to create a custom reasoner based on the *hyper-tableau algorithm* (Motik *et al.*, 2007), but that is not a high-priority task at this stage of implementation (there are other more important areas where MP implementation can be improved).

Direct reasoning on business domain models and metadata written in a general purpose language (Clojure) on a mainstream platform (Java) would yield benefits not only from the improved analysis of the created code through reasoning, but from the exposure of the approach of reasoning on software to a wider audience involved in solving real-world problems.

5.6. *Semantic Annotations in Software Engineering*

All software artifacts can be considered entities that are being annotated with semantic annotations. Compared to the annotations written in RDF/OWL, "incorporated" annotations written in the host language have the advantage of being easy to implement and use by software engineers, while still being available to translate to RDF/OWL and exposed to Semantic Web tools. Clojure's support for metadata is an excellent attachment point.

For example, the following listing uses an annotation DSL and shows a Magic Potion concept being annotated with Magic Potion statements regarding a mandatory feature (see Section 5.4):

```
(annotate policy-processor :mandatory security-test)
```

5.7. *Ontology-Driven Software Architectures*

Clojure programs created in Magic Potion are ontology-driven. They integrate semantically rich business domain models with other mainstream and/or emerging paradigms, notably functional, service-oriented, message-oriented, concurrent paradigm, etc.

The key point related to this issue is that integrating the ontology paradigm into a suitable host language makes that paradigm suitable to be an architectural cornerstone for the suitable class of applications. It also broadens the potential to be combined as a complement with the architectural aspects based on other paradigms. For example, incorporating the ontology paradigm into Clojure on Java platform via Magic Potion, makes it suitable for creating business domains models/programs that can be used for parallel computation, thanks to the good fit with the Clojure's immutable data structures and Software Transaction Memory (Djuric and Krdzavac, 2010; Djuric and Devedzic, 2010).

6. Evaluation: Is This Approach Good Enough?

Finding the best approach to solve a particular task has never been easy; there are many solutions, each of them having advantages and drawbacks. To this end, there has been a concrete experience with Magic Potion as an internalized paradigm and it is summarized here.

Since Magic Potion has been created to suit specific practical needs, it is considerably simpler and faster than heavyweight ontology solutions such as Jena. It is not a surprise that it has exactly those advantages that were defined as a goal (Section 4.1), and disadvantages that the authors were prepared to tolerate (comparative immaturity, obscurity, the need for custom maintenance etc.). However, these metrics are mostly subjective, specific to the authors and not generally relevant.

Fortunately, there has been an opportunity to evaluate this approach with a group of 30 students who attended a MSc course on Software Engineering Tools and Methodologies. Most of them were active software developers familiar with mainstream languages (Java/.NET/PHP), more than half of them were familiar with Semantic Web technology (OWL, Jena), a few of them were familiar with alternative languages (Ruby, Python) and none of them was familiar with Clojure and advanced metaprogramming. During the course, they first had to learn Clojure and then learn and use Magic Potion to create domain models for the domains they had worked with previously.

Regarding the key points that have been identified, the following has been found (summarized in Table 1):

- *Homogeneous approach*. Once the developers have learned the basics of Clojure, a majority of them (23) found Magic Potion easier to work with than external Jena repository and OWL. Almost one half of them think that Magic Potion's integration with Clojure's native constructs helped them better understand Clojure, immutable structures and the functional paradigm.

Table 1
Magic Potion evaluation summary

Issue	Advantages	Disadvantages
Homogeneous approach	– Ease of use – Helps understand Clojure	– Applicable only in host languages with rich metaprogramming support
Minimum of additional features	– Most of MP features are frequently used – Easy to learn – Helps learning Semantic Web technologies	– Does not support all features of the original paradigm
Host platform support	– Blends well with the host platform – Supported by all regular tools	– The original platform may not be preferred by some developers
Consistency	– Fully consistent with the host language	No
Clarity	– Easier to understand, learn, and use	No
Familiarity	– Intuitive to the host platform's developers	– Advantages limited to the host language
Tools and services	– Can be supported by all tools and services of the host platform – Good for prototyping and experimentation	– Applicable languages are still emerging – Not yet suitable for mainstream projects
Trends	– Advanced developers are interested in such technologies – There is a clear need – Suitable host languages gain much attention from the industry lately	– Not yet in the mainstream – Still mostly applicable for prototypes and experiments

- *Minimum of additional features*. For completing their programming tasks for the course, the majority of the developers used most of Magic Potion's features. Those who were already familiar with OWL found Magic Potion concise, focused, and very easy to learn, while those who had to learn the ontology paradigm learned it faster than the developers that were learning it through Semantic Web technologies.
- *Host platform support*: At first, developers that were used to heavyweight IDEs and graphical tools were intimidated by the spartan features in Clojure plugins for Eclipse and NetBeans. After a couple of weeks, 6 of them preferred such an environment, while the rest did not, but still found Magic Potion not requiring more than Clojure.
- *Consistency*. The majority of developers (24) found learning Magic Potion easier than learning advanced topics of Clojure. They weren't able to find inconsistencies between Magic Potion and idiomatic Clojure.
- *Clarity*. Mentally relating different platforms, metamodels, and paradigms was the most challenging task for all developers. Modeling Spaces helped, but only 10 fully

understood how Magic Potion was created and had an idea how they would do similar thing for another paradigm. Only a few understood these issues if heterogeneous approach and multiple platforms were involved.

- *Familiarity*. All developers that learned Clojure found Magic Potion very intuitive.
- *Tools and services*. All developers were concerned with how they would integrate this approach in their everyday workflows. They liked the idea and even had some ideas on their own in other domains, but were skeptical regarding its feasibility in larger projects. Those who had previous experience with dynamic languages were more enthusiastic about trying it for their experimental projects first.
- *Trends*. The developers found the concept of incorporating the ontology paradigm very compelling and in accordance with their interests. More than a half of them (18) had heard about metaprogramming, emerging alternative languages, and functional programming, and had been interested in the topic before, but the majority didn't practically pursue these interests due to time constraints and inability to find an appropriate learning path. They found the homogeneous metaprogramming approach supported by Clojure much easier than they had expected or seen in other metaprogramming tools.

7. Discussion: Who Is This For?

Magic Potion has been used in several domains of various scope (Djuric *et al.*, 2010; Djuric and Devedzic, 2010). Table 2 shows a summary of the gathered insights regarding the suitability of the ontology paradigm for the software engineering. The

Table 2
Suitability of the incorporated ontology paradigm

Objective	Advantages	Challenges
Suitability for domain modeling	Supports independent properties, complex hierarchies and multimethod-based polymorphism	Specific to the host language
Integration with Semantic Web technologies	Individuals are easily tranformable to RDF	IBased on different kind of DLs – constraint functions cannot be transformed to OWL constraints in a straightforward manner
Transactional semantics	Well-defined, STM-based, easy to use and maintain	Clojure STM – specific
Suitability for parallel computation	One of the design goals	Distributed parallelization is not supported out of the box
Formal soundness	Theoretically based on Description Logics	The implementation trades some of the purity for practicality
Developer friendliness	Concise and very readable	Clojure – specific

discussion covers Magic Potion's approach of using STM to manage immutable DL-based concept-property models with mutable references as an example of the ontology paradigm working in accord with other paradigms on the host platform (functional, concurrent, object-oriented). Not surprisingly, Magic Potion fulfills the major goals that have been defined in Section 4.1, while it has a long way to go regarding characteristics related to maturity and support.

The focus was to take a promising new language tailored for parallel programming (Clojure), and make it suitable for ontology-based and programmer-friendly domain modeling with as little intervention as possible. The main goal was to provide a concise and expressive formally sound language for domain modeling suitable for parallel programming. In this regard, Magic Potion succeeds.

Both Magic Potion and traditional approaches are suitable for domain modeling. Depending on the implementation, some of these approaches are more or less expressive, trading expressiveness for performance and easiness of use and vice-verse. Being based on DLs, MP is quite expressive, while making pragmatic choices for performance and usability. While mainstream approaches use manual locking, rendering them difficult and error-prone for parallel computation, Magic Potion employs STM to solve the concurrency problem in domain-heavy programming in a more elegant manner. As with expressiveness, most traditional languages are either simple but not very powerful of are very expressive but verbose. MP manages to be concise and very readable.

On the downside, MP approach still lacks maturity. It is expected to mature, since it is a new approach that is only in its infancy. That same infancy can also be an advantage, since it is an open canvas without the legacy baggage of older technologies. On the other end of the maturity argument, MP does not require any additional tool or methodology, just plain Clojure, so it does not impose a separate adoption cost. If Clojure is a good fit for a project and there is a need for domain modeling, MP can be seamlessly introduced. Magic Potion is still green, but fits excellently with the existing technologies, taking the best from both worlds.

It is clear that this approach is still exotic to the majority of mainstream developers. It can be a great fit for exploratory and experimental programming, prototyping and specialized solutions. The need to use multiple paradigms anyway may bring it closer to more conservative projects once the emerging languages with strong metaprogramming support that gain lots of attention of more curious and advanced developers today, also gain more ground in mainstream projects.

8. Conclusions

Homogeneous metaprogramming enables multiparadigm programming on a single platform/language. Embedding ontologies through a homogeneous DSL in a host language to support the features of the ontology paradigm that were needed is an alternative to adding a new platform to the environment. It avoids the complexity of multi-platform development. It requires less tooling, code complexity is lower while maintaining succinctness, while developers can stay inside the comfort zone of their preferred environment.

Magic Potion is a DSL meta-language for domain modeling in software development, theoretically based on description logics. It enables knowledge modeling in concurrent applications through the use of ontologies implemented in Clojure. In fact, it can be used together with Clojure to transform an ontology into an executable DSL. Yet, unlike ontology modeling languages and tools widely used in the Semantic Web community, Magic Potion is more practical from the perspective of a software developer. Its major advantages over other ontology representation languages include its concise definition and readability, as well as its well-defined transactional semantics, executability, and easiness to use and maintain.

It becomes easier than ever to experiment with different paradigms in a software team's primary software environment, tailor them to the team's needs, share and evolve them, similarly to how the programmers develop libraries and frameworks. Ontologies are a useful tool for many of the common tasks in software engineering. This work is a pragmatic and practical approach that brings the most important features of the ontology paradigm into the software engineering realm with as little disruption as possible.

References

Baader, F., Calvanese, D., McGuinness, D.L., Patel-Schneider, P., Nardi, D. (2007). *The Description Logic Handbook Theory, Implementation, and Applications*, 2nd edn. Cambridge University Press, Cambridge.

Berners-Lee, T., Hendler, J., Lassila, O. *et al.* (2001). The semantic web. *Scientific American*, 284(5), 28–37.

Djuric, D., Devedzic, V. (2010). Magic potion: incorporating new development paradigms through DSLs. *IEEE Software*, 27(5).

Djuric, D., Krdzavac, N. (2010). Software transaction memory powered domain modeling. Submitted to *Computing and Informatics*.

Djuric, D., Gasevic, D., Devedzic, V. (2006). The tao of modeling spaces. *Journal of Object Technology*, 5(8), 125–147.

Djuric, D., Jovanovic, J., Devedzic, V., Sendelj, R. (2010). Modeling ontologies as executable domain specific languages. In: *Proceedings of the 3rd Indian Software Engineering Conference* (ISEC 2010).

Fowler, M. (2004). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Gasevic, D., Djuric, D., Devedzic, V. (2009). *Model Driven Engineering and Ontology Development, 2nd edn.* Springer, New York.

Graham, P. (1993). *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Upper Saddle River.

Halloway, S. (2009). *Programming Clojure*. Pragmatic Bookshelf.

Hickey, R. (2008). The Clojure programming language. In: *Proceedings of the 2008 Symposium on Dynamic Languages*.

Hickey, R. (2011). Clojure homepage. Accessed online from: `http://www.clojure.org` on 24 June 2011.

Klyne, G., Carroll, J.J., McBride, B. (2004). Resource description framework (RDF): concepts and abstract syntax. *W3C Recommendation*, 10.

Langlois, B., Jitia, C.E., Jouenne, E. (2007). Dsl classification. In: *OOPSLA 7th Workshop on Domain Specific Modeling*.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall. Upper Saddle River.

Lavbič, D., Krisper, M. (2010). Facilitating ontology development with continuous evaluation. *Informatica*, 21(4), 533–552.

McBride, B. (2002). Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6), 55–59.

Motik, B., Shearer, R., Horrocks, I. (2007). Optimized reasoning in description logics using hypertableaux. *Lecture Notes in Computer Science*, 4603, 67.

Motik, B., Parsia, B., Hoekstra, R., Horrocks, I., Sattler, U. (2008). OWL 2 web ontology language: structural specification and functional-style syntax. *W3C Working Draft, W3C*.

Paolucci, M., Kawamura, T., Payne, T., Sycara, K. (2002). Semantic matching of web services capabilities. In: *The Semantic WebISWC 2002*, pp. 333–347.

Peng, X., Zhao, W., Xue, Y., Wu, Y. (2006). Ontology-based feature modeling and application-oriented tailoring. *Reuse of Off-the-Shelf Components*, 87–100.

Schmidt, D.C. (2006). Guest editor's introduction: model-driven engineering. *Computer*, 25–31.

Shadbolt, N., Hall, W., Berners-Lee, T. (2006). The semantic web revisited. *IEEE Intelligent Systems*, 21(3), 96–101.

Shavit, N., Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2), 99–116.

Sheard, T. (2001). Accomplishments and research challenges in meta-programming. *Lecture Notes in Computer Science*, 2196, 2–44.

Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y. (2007). Pellet: a practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2), 51–53.

Spinellis, D. (2008). Rational metaprogramming. *IEEE Software*, 78–79.

Sycara, K., Paolucci, M., Ankolekar, A., Srinivasann, N. (2003). Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1(1), 27–46.

Tsarkov, D., Horrocks, I. (2006). FaCT++ description logic reasoner: system description. *Automated Reasoning*, 292–297.

Vaira, Ž., Čaplinskas, A. (2011). Software engineering paradigm independent design problems, gof 23 design patterns, and aspect design. *Informatica*, 22(2), 289–317.

Van Deursen, A., Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6), 26–36.

Volkmann, M. (2009). Clojure – functional programming for the JVM. Accessed online from: `http://java.ociweb.com/mark/clojure/article.html` on 12. Sept 2009.

Wang, H., Li, Y.F., Sun, J., Zhang, H., Pan, J. (2005). A semantic web approach to feature modeling and verification. In *1st Workshop on Semantic Web Enabled Software Engineering* (SWESE 05), Galway.

D. Djuric is an associate professor at the University of Belgrade, Serbia. His research interests include software engineering and intelligent systems. He received his PhD in information systems from the University of Belgrade.

V. Devedzic is a professor of computer science at the University of Belgrade, Serbia. His main research interests include software engineering, intelligent systems, and applications of artificial intelligence techniques to education and healthcare. Homepage: `http://devedzic.fon.rs/`.

Ontologijos paradigmos jungimas į bendrosios paskirties programavimo aplinką

Dragan DJURIC, Vladan DEVEDZIC

Atsiradus semantiniam pasauliniam saitynui, požiūris į žinių inžineriją ir ontologijas pakito. Skirtingose paradigmose gana dažnai tenka spręsti panašias problemas ir jų sprendimo būdai taip pat gana dažnai esti panašūs. Tokios paradigmos gali abipusiai viena kitą papildyti ir patobulinti. Šiame straipsnyje pasiūlyta, kaip, jungiant ontologijas į bendrosios paskirties programavimo aplinką, jas paprastai ir lanksčiai panaudoti rašant nedideles programas. Siūlomas būdas grindžiamas metaprogramavimu, panaudojant kurį, Clojure programavimo kalba papildoma ontologinio modeliavimo paradigma. Šitaip praplėsta Clojure kalba buvo realizuota specialios paskirties programavimo kalba Magic Potion, integruojanti ontologijų, funkcinio programavimo, objektinio programavimo ir lygiagrečiųjų skaičiavimų paradigmos. Nors ir specializuota, ši kalba nėra pritaikyta kokiai nors konkrečiai dalykinei sričiai ir gali būti vartojama plačiam uždavinių ratui – pradedant e-mokymusi ir baigiant e-verslu – programuoti.