An Architecture of GALILEO: A System for Automated Ontology Evolution in Physics

An Architecture of GALILEO: A System for Automated Ontology Evolution in Physics

Michael Chan and **Alan Bundy**
School of Informatics
Unviersity of Edinburgh
{M.Chan, A.Bundy}@ed.ac.uk

Abstract

The GALILEO system implements novel mechanisms for automated ontology evolution in physics. These mechanisms, called *ontology repair plans*, resolve logical conflicts between several modular ontologies. To demonstrate that ontology evolution can be automated using ontology repair plans, we propose a flexible architecture for the implementation. Support for inference of formulae that trigger repair plans and modularisation of ontologies are central to the design. Huet's Zipper data structure is to be used to avoid encoding the object-level formalism with a deep embedding. For a high degree of modularity, the management of a collection of ontologies is handled by development graphs.

1 Introduction

Visions of applications involving multi-agent systems, e.g., the Semantic Web, have intensified the need for mechanisms for automated ontology evolution. We address the issue by developing *ontology repair plans* and applying them to records of ontology evolution in the history of physics. The implemented programs form the core of the GALILEO system (Guided Analysis of Logical Inconsistencies Leads to Evolved Ontologies). Ontology repair plans enable agents to autonomously update their own ontologies by applying diagnosis and repair operations [Bundy and Chan, 2008]. As a step towards the demonstration that automated ontology evolution via ontology repair plans is computationally feasible, the repair plans and the example ontologies are implemented in Teyjus, an implementation of λProlog [Miller and Nadathur, 1988]. Higher-order abstract syntax is used to enable the meta-logic to recurse over the syntactical structure of the object-logic. Unfortunately, this results in the inability of performing inference to reason about whether some ontology has a theorem that infers the triggering formulae of a repair plan. One consequence is that theorems are forced to be coded as axioms. This implementation is clearly unnatural, but has been kept simple because it is a mere proof-of-concept. A more flexible architecture is essential in order to go beyond.

The goals underlying the design of the architecture of the new implementation are two-fold:

- Higher-order reasoning engines must be incorporated to infer formulae that trigger ontology repair plans.
- The representation of knowledge as ontologies should be modular.

In the current pure λProlog implementation, the inference is left to the programmer and the encoding of the ontologies for repair is tailored to the specific triggering formulae of the repair plans. This issue can be addressed by adopting a reasoner, such as a theorem prover. Only higher-order reasoners are useful to us because many physics properties are more realistically represented as higher-order objects. For instance, a star's orbit is more realistically represented as a function object that returns the 3-D spatial position of the star for a given time moment. If it were represented as a non-functional object, then much of the description and expressivity would be lost and the representation would be much closer to that of static, physical entities, e.g., a dog or a house.

The idea of making the knowledge-base modular has received much attention from the knowledge representation community. For instance, it is a basis of Sowa's notion of "knowledge soup" [Sowa, 2006]; it suggests that because the knowledge in a person's head is of a disorganised and dynamically changing nature, it consists of many self-consistent chunks of knowledge. Similarly in Cyc [Lenat, 1995], different contexts are represented as separate *microtheories*; each is represented by a set of assertions that corresponds to the relevant facts and assumptions valid in the particular context. Our current representation is modular as the predictive and observational ontologies are already separate representations [Bundy, 2008]. Each of these is internally consistent, but possibly inconsistent with one another [Bundy and Chan, 2008]. We will extend this principle by modularising other contents as well, e.g., foundational theories of physics and mathematics.

2 Reasoning for Ontology Evolution

Physics ontologies are represented at the object-level, quantifying over physics functions, objects, and measurements. Object-level reasoning is therefore needed to, e.g., derive the Law of Universal Gravitational Force from an ontology containing the axioms of the Newtonian theory. Repair plans are designed to guide the ontology repair procedure, so they need to be represented at the meta-level quantifying over objects at

the object-level. Meta-level reasoning is therefore clearly distinct from that of object-level; for instance, it reasons about whether an ontology can trigger a repair plan by matching some of its theorems to the trigger formulae. Moreover, ontology manipulation is also guided by the meta-level, e.g., the change of signatures and the addition of axioms.

Some higher-order theorem provers, such as Isabelle [Nipkow *et al.*, 2002], offer a clear separation of the logic into meta- and object-levels. We shall discuss the possibility of using a theorem prover as the sole basis for the implementation.

2.1 Effects of Shallow and Deep Embeddings

There are generally two techniques of encodings: shallow and deep embeddings. In a shallow embedding, only the semantics of the logic needs to be defined. As the syntactic structures are not represented, theorems about the syntax cannot be proven. If the two logics are both shallowly-embedded, it is then difficult for the meta-logic to reason about the syntactical structure of the object-level formalism, e.g., matching a theorem to some trigger formulae. Syntax then cannot be deconstructed by pattern matching because higher-order unification attempts to instantiate some variable to the constant in question. We do not want a variable to be instantiated when almost any constant is pattern matched against it.

In a deep embedding, both the (abstract) syntax and the semantics of the object-logic must be defined. The syntax is typically defined by an inductive definition. In effect, statements of an object-logic are represented as objects of a data type of the meta-logic. For instance, $f(x)$ becomes `(applic f x)`, where `applic` is of type `(A -> B) -> A -> B` and both `f` and `x` are constants. If inference is to be enabled, the unification algorithm may even need to be reimplemented. Note that the current λProlog implementation uses deep embedding to represent the two logics as one.

Neither a pure shallow nor deep embedding appears to enable both inference and the matching and instantiation of trigger formulae. Our solution involves a kind-of hybrid approach.

2.2 Huet's Zipper Data Structure

Due to the rigorous formalisation required, the effort for producing a deep embedding that allows inference to be done can be substantial. Our solution aims at the accommodation of both the shallowly-embedded formulae and their parse trees. It is based on using Huet's *zipper* data structure [Huet, 1997], a data structure that can deconstruct any list- and tree-like data structure and requires a constant time to move in any location.

The creation of a zipper from the parse tree of a formula resembles the translation from shallow to deep embedding. Of course, it is not true deep embedding since the zipper does not carry information about the semantics, but the representation of the parse tree in a zipper is sufficient for our needs. Navigating around a zipper naturally corresponds to moving around terms, so operations for term manipulation can then become intuitive. So, we will have the best of both shallow and deep embeddings by passing around both the formula and the zippers. The formula can be used for inference, while the zipper containing the parse tree can be used for pattern matching.

3 Modularising Knowledge

Currently, the theoretical and sensory ontologies are treated as separate interacting logic theories, e.g., the *Inconstancy* ontology repair plan resolves conflicts between a theoretical and multiple sensory ontologies [Chan and Bundy, 2008]. A major theoretic advantage of having multiple, internally consistent ontologies is the better management of logical inconsistency. Since inconsistent theories can prove all formulae, all instantiations of the triggers of all repair plans would also be provable. Keeping the theories consistent only internally avoids the combinatorial explosion of instantiations, yet a conflict can still be detected at the global level. A high degree of modularity also makes the integration of new ontologies more natural and convenient. Instead of adding axioms and changing type declarations to existing ontologies, which can invalidate previous proofs, a new ontology can be wholly introduced to the collection. This is in fact close to a central principle of object-oriented programming; that is, the workings of an object are decoupled from the rest of the system and the addition of a class does not affect other parts of the program.

To further modularise the existing knowledge representation, the physics and mathematical theories can be partitioned into small ontologies. Each ontology resembles a small context, e.g., there could be separate ontologies for a formal theory and for a naive theory of geometry, which only covers 2-D spaces. Certainty factors can be assigned to an ontology, determining the vulnerability of the theory or experiment to repair. For instance, the ontology for a controversial theory should be valued at a lower confidence than an established one. The factors could be expressed using a discrete representation, e.g., `definitely true` and `true by default`, as experimented in Cyc.

3.1 Partition Management with Development Graph

The collection of ontologies can be translated to form a structured logical representation called a *development graph* [Autexier *et al.*, 1999], in which a node corresponds to an ontology. A node, thus an ontology, can be defined to import signatures and theorems from other nodes. Such morphisms between theories are formalised using *definition links*. To help separate the axiomatisation of the theory itself from the imported, the axiomatisation is split into local and global parts.

For the implementation of ontology repair, repaired ontologies are inserted into the development graph as new nodes, replacing those that correspond to the original ontologies. The new nodes should have the same morphisms as the old, so the morphisms of the replaced nodes are preserved. Clearly, some of the old morphisms may no longer be desirable depending on the type of repair performed and the notion of minimal change adopted. The new nodes could partly import the axiomatisation of the replaced, giving them implicit definitions. In this case, the replaced nodes still exist, but are disconnected from the rest of the system.

Development graph is already implemented in HETS [Mossakowski *et al.*, 2007] and MAYA [Autexier *et al.*, 2002], which are systems for managing evolutionary development and for analysis and proof management, respectively. In both systems, development graph can be created by translating an input HASCASL specification, a higher-order extension of CASL.

3.2 Example Specification

[debugging]

4 Putting the Pieces Together

To leverage an existing implementation of development graph, we are currently experimenting with HETS because the implementation is better maintained. Unfortunately, HETS has no support for evolutionary change management. We will likely migrate to a system to be called DocTip, the successor of HETS and MAYA, once it becomes available. Both DocTip and HETS can be connected to a range of theorem provers, including Isabelle. So, the trigger formulae of a repair plan can be inferred from the conflicting ontologies if the ontologies are encoded by using a shallow embedding.

The repair plan programs are responsible for directing the modification of the development graph, e.g., adding and deleting nodes and morphisms. These therefore need to communicate to the development graph editor via an API. A polymorphic, higher-order logic programming language, e.g., λProlog, is needed to encode trigger formulae and transformation rules.

It is, however, not yet clear how to correctly instantiate the trigger formulae by recursing over the syntactical structure of the object-level formulae. Future work will involve the investigation of the use of zippers in HETS or DocTip.

5 Conclusion

The current λProlog implementation deeply-embeds the meta- and object-logics into one, and this approach limits the ability to perform inference on the trigger formulae. As an alternative, we have described the possibility of using Huet's Zipper data structure to encode the parse tree of a formula. Our new architecture accommodates the use of various reasoning engines, e.g., Isabelle, to prove theorems of ontologies. Since partitioning the knowledge-base into small ontologies help control logical inconsistency and provides easier maintenance, we go beyond just separating the predictive and sensory ontologies. Development graph is used for the management of the highly modularised ontologies.

References

[Autexier *et al.*, 1999] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. *Lecture Notes In Computer Science; Vol. 1827*, pages 73–88, 1999.

[Autexier *et al.*, 2002] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 495–501. Springer-Verlag London, UK, 2002.

[Bundy and Chan, 2008] A. Bundy and M. Chan. Towards ontology evolution in physics. In W. Hodges, editor, *Procs. Wollic 2008*. LNCS, Springer-Verlag, July 2008.

[Bundy, 2008] Alan Bundy. *Automating Signature Evolution in Logical Theories*, volume 5144/2008, pages 333–338. Springer Berlin / Heidelberg, Jul 2008.

[Chan and Bundy, 2008] M. Chan and A. Bundy. Inconstancy: An ontology repair plan for adding hidden variables. In S. Bringsjord and A. Shilliday, editors, *Symposium on Automated Scientific Discovery*, number FS-08-03 in Technical Report, pages 10–17. AAAI Press, November 2008. ISBN 978-1-57735-395-9.

[Huet, 1997] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[Lenat, 1995] D.B. Lenat. CYC: a large-scale investment in knowledge infrastructure. *Commun. ACM*, 38(11):33–38, November 1995.

[Miller and Nadathur, 1988] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*. MIT Press, 1988.

[Mossakowski *et al.*, 2007] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Sowa, 2006] J.F. Sowa. The challenge of knowledge soup. In J. Ramadas and S. Chunawala, editors, *Research Trends in Science, Technology and Mathematics Education*, Mumbai, 2006. Homi Bhabha Centre.