THE JESSTAB APPROACH TO PROTÉGÉ AND JESS INTEGRATION

Henrik Eriksson

Department of Computer and Information Science

Linköping University

SE-581 83 Linköping, Sweden

her@ida.liu.se

Abstract Communication with performance systems is becoming increasingly important for ontology development and knowledge-modeling tools. Such tools can benefit from tight integration with inference engines. JessTab is an extension to the Protégé framework that allows developers to take advantage of the Jess performance system. JessTab supports declarative mapping of sections of Protégé knowledge bases to the Jess knowledge representation and adds functionality for operations on ontologies and knowledge bases.

Keywords: Knowledge acquisition, knowledge engineering, ontologies, expert-system shells

1. Introduction

Development environments for ontology definition and knowledge modeling require integration with other types of software. For example, it is important to communicate with other programs and to control external programs. Furthermore, it is sometimes desirable for external programs to control a development environment (e.g., to open an ontology editor and to focus on a particular part of the ontology). The ability to integrate development environments with other software is a critical factor for using these systems in a wide range of organizations and a key to widespread acceptance of such environments.

The Protégé (Grosso et al., 1999) development tool provides an application programmer interface (API) for this type of integration. However, this API is based on low-level concepts. It allows calls to and from the environment implementation language (Java). There is no support for

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35602-0_35](https://doi.org/10.1007/978-0-387-35602-0_35)

integration with declarative languages and there are no shells suitable for implementation of problem solvers.

Protégé is essentially a stand-alone tool in the sense that it produces output (i.e., ontologies and knowledge bases) that other programs can read. Protégé assumes that the performance engine runs as a separate program, perhaps on a different platform. This separation is advantageous in most situations because the development and run-time environments are typically different. However, there are several situations in which tight integration with a performance system would be useful.

In many situations it is useful to create programs that perform operations on ontologies or modify the knowledge base in certain ways. A developer might want to define rules that match patterns in the knowledge base and trigger actions, such as filling in information derived from other values and making adjustments that refine the knowledge base. Alternatively, a developer might want to run problem-solving methods in the Protégé environment to enable immediate testing.

JessTab is an approach to integrate two popular systems—Jess and Protégé. The JessTab implementation allows developers to create knowledge bases in Protégé that work together with Jess programs and rule bases. It creates a declarative mapping between the knowledge representation schemes supported by these tools. This mapping unifies the Protégé and Jess formats by mapping sections of the knowledge base to the Jess fact base. JessTab can propagate automatically modifications to mapped Protégé instances on Jess.

We found that having an integrated performance component in a tool such as Protégé has several advantages. Developers can use the performance component as a macro language for manipulating of knowledge bases. Rules and functions can automate repetitive tasks (e.g., during maintenance), and can ensure consistency. Furthermore, it is possible to use JessTab as an object-oriented extension to Jess by using the ontology representation supported by Protégé. In this approach, Protégé is an extension to Jess.

In the remainder of this paper, we shall describe the background in terms of Protégé and Jess, and present the JessTab approach in detail. Finally, we discuss the potential uses of JessTab and considers directions for future research in this area.

2. Background

Before discussing the JessTab extension, let us first examine the current Protégé-2000 implementation, the axiom-language extension to Pro-

tégé (which is a complementary extension), and the Jess language and its implementation.

2.1. Protégé-2000

Protégé-2000 is a tool that supports knowledge acquisition and the development of knowledge bases (Grosso et al., 1999). It includes an ontology editor and a system for generating and custom tailoring forms for data entry by domain specialists. The Protégé-2000 user interface consists of several *tabs* that provide different views of the knowledge base. For example, there are tabs for ontology editing, form layout, knowledge entry, and knowledge-base search. Protégé-2000 is implemented as an open-source Java application.

An important feature of Protégé-2000 is the possibility to extend its functionality through plug-in modules. By defining *slot-widget* plug-ins, developers can extend the functionality of data-entry widgets (e.g., by implementing new interaction modes). The mechanism for *tab-widget* plug-ins allows developers to add new tabs to the Protégé-2000 user interface. Developers often use this plug-in API to implement project-specific tabs that provide unique views of the knowledge base.

2.2. PAL: The Protégé Axiom Language

PAL is a language for expressing constraints in Protégé knowledge bases. It is based on first-order predicate logic. PAL allows developers to define constraints that can be checked by the PAL interpreter. Once the knowledge base under development is in a reasonably coherent state, the developer can invoke the PAL engine to check the consistency with respect to the constraints. PAL then reports any constraint violations to the developer. PAL is implemented as a tab-widget plug-in to Protégé.

The main difference between JessTab and PAL is that JessTab allows developers to use Jess as a programming language and inference engine. PAL, however, is not a programming language in the sense that it can be used to define inferences and behaviors. Developers invoke PAL explicitly to perform constraint checking. The result of this process is a report (rather than a behavior). Unlike Jess, PAL is custom tailored for expressing constraints for checking knowledge bases.

Jess and PAL are complementary languages in the sense that PAL can be used to test the integrity of a knowledge base, whereas Jess can be used to implement operations on the knowledge base. Because Jess is Turing complete, it is possible to use it to implement custom-tailored constraint checkers as well as programs that modify the knowledge base.

Figure 1. Jess console window as a Protégé-2000 tab.

2.3. Jess

Jess (Friedman-Hill, 1997; Watson, 1997) is a reimplementa-tion of a subset of CLIPS (NASA, 1991) in Java. Jess implements some additional functionality not provided by CLIPS. Like CLIPS, reasoning in Jess is based on a list of known *facts* and a set of *rules* that match on these facts in its fact base.

Facts are basically a list of known values, such as (`temperature outdoor 25`), and the preconditions of the rules are patterns with wild-cards and conditional expressions, such as (`temperature ? ?t&:(> ?t 20)`). Rules can assert new facts when they fire, which in turn could re-sult in other rules firing. Jess uses the Rete algorithm to ensure efficient pattern matching and rule activation (Forgy, 1982). Jess supports both forward chaining and backward chaining.

3. The JessTab Approach

JessTab provides a Jess console window in the Protégé environment (see Figure 1). Developers can easily switch between the Jess read-evaluate-print dialogue and the normal graphical interaction with Pro-tégé. To make Jess development in the Protégé environment easier, JessTab implements graphical inspectors for Jess facts and definitions (rules, functions, defglobals, deftemplates, and deffacts).

In addition to the graphical tools, JessTab extends the Jess and Protégé functionality in five important areas:

- 1 Mapping instances to facts. JessTab maps Protégé knowledge bases consisting of instances to facts in the Jess fact list.
- 2 Functions for knowledge-base operations. JessTab adds several functions to Jess that allow Jess programs to query and manipulate Protégé knowledge bases.
- 3 Functions for ontology reflection and operations. JessTab adds functions that examine and modify the Protégé ontology.
- 4 Mirroring Jess definitions in Protégé knowledge bases. Instances represent Jess definitions, such as rules and functions.
- 5 Support for methods and message handlers in Jess. Currently under development.

The remainder of Section 3 covers basic integration of Jess with Protégé instances (Areas 1–2) while Section 4 discusses issues related to ontology integration (Areas 3–4).

3.1. Mapping Protégé Knowledge Bases to Jess Facts

A core component of Protégé–Jess integration is the *mapping* mechanism. This approach makes it possible to mark classes in the ontology for mapping to Jess facts. By using the expression (`mapclass <class-name>`), developers can map all instances of the specified class to Jess facts. The `mapclass` expression applies to direct as well as indirect instances (i.e., instances of subclasses). An alternative to the class-mapping mechanism is to map a specific instance to a fact through the expression (`mapinstance <instance>`). An important design goal for the JessTab mapping mechanism is to enable Jess rules to pattern match on mapped instances. We can then write rules that fire when the appropriate conditions in the Protégé knowledge base are met.

Let us consider an example. Mapping the class `Person`, which has a single instance representing “John” age 20, results in a Jess fact with the format (`object (is-a Person) (name "John") (age 20)`). This fact represents the instance with its slot values for `name` and `age`.¹

¹The fact in this example is simplified for clarity; the actual fact holds additional information about the instance.

JessTab asserts facts based on a certain pattern (template). The pattern starts with `object` and it always contains the class name and a direct reference to the Protégé object. As shown in the example, the slot-value pairs of the instance are also part of the fact. The mapping mechanism asserts a fact for every instance (of a mapped class). Creating a new instance of the class `Person`, for example, will result in the assertion of a new fact matching the instance.

Our motivation for designing this type of mapping to facts is that it enables developers to define Jess rules that pattern match on mapped instances in many ways. For example, it is possible to write rules that match on instances regardless of their class by using a wildcard for the `is-a` property. The pattern `(object (is-a ?) (name "John"))` matches every (mapped) instance that has a slot *name* with the value "John".

The mapping from instances to facts is unidirectional in the sense that changes to the Jess fact list, such as assertions of facts, are *not* propagated back to the Protégé knowledge base. However, it is possible to make changes to the knowledge base from Jess by calling functions that operate on knowledge-base constructs.

3.2. Knowledge Base Operations from Jess

JessTab provides several functions for retrieving information from Protégé knowledge bases and for modifying them. There are functions for getting slot values from instances, for defining new classes, and for instantiating classes. Whenever possible, these functions are compatible with their corresponding CLIPS counterparts (if any).

Let us consider how we can use JessTab to create Protégé classes and instances from Jess. Just as in CLIPS, developers can use the `defclass` construct² to create a new class and attach slots to it. The `make-instance` function allows us to create an instance of a specified class in the knowledge base and, optionally, provide initial slot values for it. Using `defclass` and `make-instance` we can now experiment with creating instances and then map them to facts. The following example illustrates how we can create a class `Person` and create a person instance named "John":

```
Jess> (defclass Person (is-a :THING)
      (slot name (type string))
      (slot age (type integer)))
```

²Jess also implements an alternative format for `defclass` (Friedman-Hill, 1997). Advanced developers can use this format to specify communication with native Java classes.

TRUE

```
Jess> (make-instance john of Person (name "John") (age 20))
```

```
<External-Address:SimpleInstance>
```

```
Jess> (mapclass Person)
```

```
Person
```

The `mapclass` expression marks the `Person` class as mapped, and it will assert facts of the (direct and indirect) instances of `Person`. After the `mapclass` operation, we can check the resulting Jess fact:

```
Jess> (facts)
```

```
f-0 (object (is-a Person) (is-a-name "Person")
          (OBJECT <External-Address:SimpleInstance>
                (age 20) (name "John"))
```

```
For a total of 1 facts.
```

This fact represents the instance of the class `Person` we created. In addition to the class name and the slot-value pairs, the fact contains a reference to the actual Protégé instance (the Java object representing it). This reference is useful when defining rules that perform operations on the object in the action part. The `is-a-name` value is a string version of the class name, which is useful in some rule patterns.

Because the class `Person` is marked as mapped, changing a Protégé slot value results in a change of the corresponding fact. We use the `slot-set` function to change the age to 21:

```
Jess> (slot-set john age 21)
```

```
Jess> (facts)
```

```
f-1 (object (is-a Person) (is-a-name "Person")
          (OBJECT <External-Address:SimpleInstance>
                (age 21) (name "John"))
```

```
For a total of 1 facts.
```

When we change a fact property, Jess actually retracts the old fact and asserts a new version of it (i.e., the fact number changes). This behavior is normal Jess semantics.

Creating a second instance adds a new fact to the Jess list of facts:

```
Jess> (make-instance sue of Person (name "Sue") (age 22))
```

```
<External-Address:SimpleInstance>
```

```
Jess> (facts)
```

```
f-1 (object (is-a Person) (is-a-name "Person")
          (OBJECT <External-Address:SimpleInstance>
                (age 21) (name "John"))
```

```
f-4 (object (is-a Person) (is-a-name "Person")
          (OBJECT <External-Address:SimpleInstance>
                (age 22) (name "Sue"))
```

```
For a total of 2 facts.
```

Because the mapping of the class `Person` is reactive, JessTab asserts a new fact automatically when we create a new instance.

We can now write a Jess rule that pattern matches on the facts. The following rule prints the name for every person of the age 21 or older:

```
Jess> (defrule twentyone
      (object (is-a Person)
        (name ?n) (age ?a&:(>= ?a 21)))
  =>
      (printout t "The person " ?n
        " is 21 or older" crlf))
TRUE
Jess> (run)
The person John is 21 or older
The person Sue is 21 or older
2
Jess>
```

This rule format allows us to write rules that perform object matching in several different ways. For example, we can write rules with constraints or wildcards for the class name. The action part of the rules can perform reasoning tasks, such as asserting new facts and making changes to slot values. Note that the format of these rules is similar to the CLIPS rule format for pattern matching on objects (as of CLIPS version 6.0).

To perform active modification of the knowledge base, we need functions for accessing and modifying values in Protégé. The functions `slot-get` and `slot-set` allow us to retrieve and change slot values, respectively. For example, the expression `(slot-get sue age)` will return the age slot value for the instance `sue`. The expression `(slot-set sue age 23)` will set the slot value for `age` to 23. Continuing the previous example, we can define a generalized rule that changes the age from 22 to 23 for every person:

```
Jess> (defrule changeage
      ?o <- (object (is-a Person) (age 22))
  =>
      (slot-set ?o age 23))
TRUE
```

There are also functions for getting and setting facet values (i.e., slot properties). You can use expressions `(facet-get <class-name> <slot-name> <facet-name>)` and `(facet-set <class-name> <slot-name> <facet-name> <new-value>)` to access the facets.

4. Ontology Integration

JessTab allows Jess programs to integrate with Protégé ontologies in a fundamental way. There are several ways in which Jess programs can examine and manipulate ontologies. Let us consider four aspects briefly:

- 1 **Ontology reflection.** It is possible for Jess programs to reflect on the makeup of Protégé ontologies. Built-in functions, such as `class-subclasses`, can discover the structure of ontologies. These functions enable us to develop Jess programs that analyze ontologies.
- 2 **Meta-level objects.** Protégé provides advanced support for meta-level objects (Eriksson et al., 1999). For example, classes are instances of metaclasses. Slots and facets are instances as well. JessTab supports these meta-level objects by allowing programs to create new classes by instantiating a metaclass (e.g., with the `make-instance` function).
- 3 **Mapping metaclasses.** Because classes behave as instances (of a metaclass) in most aspects, it is possible to map them to facts just as regular instances. Thus, by mapping classes to facts, we can apply Jess rules to Protégé classes. For example, it is possible to write rules that pattern match certain classes in a large ontology and modify them.
- 4 **Definition mirroring.** JessTab mirrors Jess definitions in the Protégé ontology. For example, Jess defrules are mirrored as an instance of the system class :`JESS-RULE` in Protégé. These mirror instances contain a textual representation of the Jess definition (i.e., the source code). Thus, Jess programs can reflect the structure of themselves by inspecting the mirror instances (e.g., by mapping them to facts and applying rules). Furthermore, it is possible to save these mirror instances together with the rest of the knowledge base, and to reinsert them during reload from files.

In summary, the meta-level objects in Protégé fit well together with Jess.³ JessTab enables Jess programs to inspect and modify ontologies just like instances, which result in a superior expressive power for rules operating on the ontology.

5. Discussion

During its development, the scope of Protégé has been broadened from an environment that mainly supported knowledge acquisition from domain experts through generating knowledge-acquisition tools to a general tool for ontology and knowledge base development with an open architecture (Eriksson and Musen, 1993; Grosso et al., 1999). The advan-

³Note that CLIPS does not support metaclasses or other meta-level objects at all.

tage of today's Protégé is the combined power of the plug-in components and the possibility to custom tailor the environment by developing your own components.

A practical goal for JessTab is that it should be a useful tool for projects originating from both the Protégé and Jess communities. For the Protégé user, the JessTab extension adds a performance engine to Protégé and support for a declarative mapping between Protégé and Jess to this tool-box. For the the Jess developer, JessTab makes it possible to take advantage of Protégé as a graphical development environment and to use third-party plug-ins (tabs) added to Protégé.

A development tool and a performance system working together is a powerful combination, and one which developers can use in several ways. The approach of combining graphical environments and reasoners is not new. In the mid-1980s, hybrid tools such as KEE featured graphical development environments (Fikes and Kehler, 1985). These tools, however, did not have the richness and flexibility of current knowledge-modeling tools and they lacked the suppleness of the Java-based Jess engine. JessTab enables developers to use the performance system for ontology and knowledge base development (e.g., by using Jess to query and modify ontologies). There are also recent tools that have performance counterparts, such as OntoEdit (which uses Ontobroker (Decker et al., 1999)), WebODE (Arpírez et al., 2001), WebOnto (Domingue, 1998), and Ontosaurus (Swartout et al., 1996) (which uses Loom (MacGregor, 1991)).

The current JessTab implementation opens interesting possibilities for future development. One potential extension is to make Protégé a Jess development environment by taking advantage of the graphical capabilities of Protégé. This approach would require a tight integration between graphical editing in Protégé and Jess definitions and structures. Some JessTab users have requested a different mapping between Protégé instances and Jess facts. A generalized mapping mechanism that allows users to define their own mappings, perhaps as Jess rules, would permit custom `mapclass` and `mapinstance` operations. An extension currently under development is support for CLIPS-like message handlers and methods. This approach will bring Jess and JessTab closer to a full object system.

One major advantage of the Protégé-Jess combination is that developers can use other Protégé plug-ins and other Jess extensions and interfaces. For example, Jess and JessTab enable developers to use Protégé for development of software agents. The Java Agent Development Framework (JADE) (Bellifemine et al., 2001) can communicate with Jess, which in turn communicates with Protégé using JessTab. Using this approach, developers can get Protégé ontology support for JADE

agents by using Jess rules. Jess rules can create Protégé instances based on messages from agents. Furthermore, Jess rules can create and send agent messages based on the content of Protégé knowledge bases.

Another possibility is to use Protégé and Jess for the development of semantic-web applications (Berners-Lee et al., 2001). Protégé supports encoding of ontologies in the resource description framework (RDF) format (Lassila, 1998). Thus, Jess can operate on such ontologies through JessTab. Used in this fashion, the Protégé–Jess combination is a building block for the semantic web.

6. Conclusion

JessTab allows developers to write Jess programs that manage Protégé ontologies and knowledge bases. Furthermore, the Jess inference engine enables developers to apply rule-based reasoning to Protégé definitions. The Protégé meta-object protocol makes it possible to apply this reasoning to meta-level objects and to use JessTab to make structural changes and extensions to classes, slots, and facets.

The JessTab project contributes to the understanding of how ontology editors and performance systems can work together. This understanding includes the mapping from the Protégé knowledge representation to the Jess representation and the use of Jess definitions as system classes in Protégé ontologies. In practice, JessTab contributes to the Protégé community by enabling the use of Jess under Protégé. JessTab also supports the Jess community by allowing the use of Protégé as an object-oriented extension to Jess and by providing a tool for using Protégé as a graphical front-end to Jess.

JessTab is a solid basis for further development of the Protégé–Jess integration. Examples of such potential integration projects are the representation of Jess problem solvers in ontologies and detailed modeling of Jess definitions (e.g., rules) in knowledge bases. Furthermore, the JessTab approach is a road map for integrating Protégé and other performance systems.

Acknowledgments

The JessTab user community contributed with testing, bug reports, and suggestions for further improvements. Ray Ferguson and Ernest Friedman-Hill provided Protégé-2000 and Jess support, respectively. Samson Tu provided valuable insights in the problem of transferring information from Protégé-2000 and to Jess via Java.

References

- Arpírez, J. C., Corcho, O., Fernández-López, M., and Gómez-Pérez, A. (2001). Web-ODE: A workbench for ontological engineering. In *Proceedings of K-CAP 2001*, Victoria, B.C., Canada.
- Bellifemine, F., Poggi, A., and Rimassa, G. (2001). Developing multi-agent systems with a FIPA-compliant agent framework. *Software: Practice and Experience*, 31(2):103–128.
- Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–44.
- Decker, S., Erdmann, M., Fensel, D., and Studer, R. (1999). Ontobroker: Ontology based access to distributed and semi-structured information. In *Proceedings of DS-8: Semantic Issues in Multimedia Systems*, pages 351–369.
- Domingue, J. (1998). Tadzebao and WebOnto: Discussing, browsing, and editing ontologies on the web. In *Proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada. Also available as <http://ksi.cpsc.ucalgary.ca/KAW/KAW98/domingue/>.
- Eriksson, H., Fergerson, R. W., Shahar, Y., and Musen, M. A. (1999). Automatic generation of ontology editors. In *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling, and Management*, pages 4.6.1–4.6.20, Banff, Canada. Also available as <http://sern.ucalgary.ca/KSI/KAW/KAW99/papers/Eriksson1/>.
- Eriksson, H. and Musen, M. A. (1993). Metatools for knowledge acquisition. *IEEE Software*, 10(3):23–29.
- Fikes, R. and Kehler, T. (1985). The role of frame-based representation in reasoning. *Communications of the ACM*, 29(9):904–920.
- Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37.
- Friedman-Hill, E. J. (1997). Jess, the Java expert system shell. Technical Report SAND98-8206, Sandia National Laboratories. Also available as <http://www.prod.sandia.gov/cgi-bin/techlib/access-control.pl/1998/98820/%6-r.pdf>.
- Grosso, W. E., Eriksson, H., Fergerson, R. W., Gennari, J. H., Tu, S. W., and Musen, M. A. (1999). Knowledge modeling at the millennium (The design and evolution of Protégé-2000). In *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling, and Management*, Banff, Canada. Also available as <http://sern.ucalgary.ca/KSI/KAW/KAW99/papers/Grosso1/kmatm.pdf>.
- Lassila, O. (1998). Web metadata: A matter of semantics. *IEEE Internet Computing*, 2(4):30–37.
- MacGregor, R. (1991). Inside the LOOM classifier. *SIGART Bulletin*, 2(3):70–76.
- NASA (1991). *CLIPS Reference Manual*. Software Technology Branch, Lyndon B. Johnson Space Center, NASA, Houston, TX.
- Swartout, B., Patil, R., Knight, K., and Russ, T. (1996). Toward distributed use of large-scale ontologies. In *Proceedings of the Tenth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 32.1–32.19, Banff, Canada. Also available as http://ksi.cpsc.ucalgary.ca/KAW/KAW96/swartout/Banff_96_final_2.html.
- Watson, M. (1997). *Intelligent Java Applications for the Internet and Intranets*, chapter 8. Morgan Kaufmann Publishers, San Francisco, CA.