ifgi
Institute for Geoinformatics
University of Münster

FLUMAGIS

How can a knowledge base run executables on the frame level?

Reiner Borchert

Reiner Borchert
FLUMAGIS project
Institute for Geoinformatics
University of Münster
Robert-Koch-Str. 26-28
D-48149 Münster
Germany
reiner.borchert@uni-muenster.de
http://www.flumagis.de

Summary

Conventional Ontologies are static constructs; they don't support execution of actions or functions, even if they describe dynamic domains like task ontologies or process ontologies. On the other side executable modules could easily be created to represent functions, actions, and more complex processes. In order to benefit from both worlds a programming language could be used which provides ontologies as well. This paper proposes a different approach to this problem: the combination of well known and widespread ontology and programming languages. In this case well defined interfaces between the ontology and the executables are essential.

In order to show how ontological frames and executables can communicate this paper introduces a basic framework which links external Java classes to frames of the knowledge base editor Protégé 2000. It turned out that object oriented programming provides a good chance to establish executable modules in ontologies.

Three fundamental operation types have been identified:

- *functions* calculate the values of certain slots on a reading access to the function slot value.
- *constraints* check new slot values on a writing access to the associated slot; can block the access if the values violate certain rules.
- *actions* are the most general type; must be triggered (usually by the user), can do then anything, e.g. change some slot values.

1 Introduction

While setting up a knowledge base concerning the water domain in the FLUMAGIS project (ecology, hydrology, etc.) we realized that there is a lack of functionality in *Protégé-2000* compared to spread sheet software (e.g. *Microsoft Excel* or *Lotus 1-2-3*):

1. It would be useful to evaluate the slot values of a certain instance and update the result automatically when a slot value has been changed, e.g. calculating the average of numeric values or concatenating parts of a name to a whole (e.g. genus name: "*Fagus*", species name: "*sylvatica*", whole name: "*Fagus sylvatica*" [= beech tree]). By the way, composed names like these could often serve very well as the displayed instance names rather than values of a single slot.
2. Also, we missed the implementation of constraints that would refuse invalid inputs immediately, or that would allow restrictions on allowed slot values that exceed the built-in constraints ("value type", "allowed classes", "cardinality").
3. Since we wanted to run specific and complex calculations and simulations on the knowledge base, we were looking for possibilities to include executables that a user can start (e.g. by clicking on a button) and which would be able to change the content of the knowledge base while running.

So we were looking for tools or plug-ins which could provide these issues, and we found some interesting rule engine approaches like *CLIPS*, *Jess*, *Jade*, and *Algernon*. All of them are available as *tab widget plug-ins* for *Protégé*.

After all it turned out that none of them could cover the entire desired functionality, although they might be partially useful. As the plug-ins appear as tab widgets, they work only "on demand" (after

activating and focusing their tab). Primarily rule engines are inference and query machines, even if they also can modify the knowledge base. The rules are applied on the entire knowledge base, rather than on a certain class or slot, regardless of specific scopes of rules. The different approaches deal each with slightly different logical languages in order to express rules, queries, and constraints.

Considering the complexity of calculations and evaluations of our project we did not get the impression that we would be able to implement them with one of the rule engines. So we thought about using an object-oriented programming language, which would provide some benefits:

- a better performance, since the rules don't need to be parsed and interpreted,
- a more efficient integration in the host program *Protégé* (e.g. by developing slot widgets to be applied in normal forms instead of separate tab widgets),
- the chance to program just anything, not limited by constraints of a rule engine.

On the other side rules will lose evidence for domain experts if they are expressed in Java rather than in a logic language.

1.1 Dynamic Ontologies?

Conventional Ontologies are static constructs; they don't support execution of actions or functions, even when they describe dynamic domains like task ontologies or process ontologies.

Executable modules could easily be created to represent functions, actions, and more complex processes. (Raubal and Kuhn forthcoming 2004) showed that a programming language (*Haskell*) can be used for ontologies and simulated processes as well. (Gaio, Lopes et al. 2003) proposed an extended DAML-S to describe and execute conditions and actions. (Mota, Botelho et al. 2003) introduced an "Object Oriented Ontology Framework" based on DAML+OIL and other languages. The approach of (Freire and Botelho 2002) "enables agents to execute any interaction protocol that can be expressed in the proposed XML representation". They used a converter to create Java classes from XML schemas.

This paper proposes a different approach to this problem: the combination of a widespread ontology editor and an object-oriented programming language. In this case well defined interfaces between the ontology and the executables are essential.

The benefit of this approach is obvious: In order to create knowledge bases one can use a technically mature and comfortable frame editor (Protégé). All executable features can be developed in a common programming language (Java), focusing on their special tasks since a complete frame infrastructure including an expandable user interface is already available.

In order to show how ontological frames and executables can communicate with one another this paper introduces a basic framework which links external Java classes to frames of the knowledge base editor Protégé-2000 (Noy, Fergerson et al. 2000). It turned out that object oriented

programming provides a good chance to establish executable modules in ontologies, provided that the ontology editor supports a broad *application programming interface* (API) like Protégé does.

1.2 The Protégé Knowledge Model

Protégé is an OKBC-compatible frame-based tool for editing ontologies and acquiring knowledge. All elements of a Protégé knowledge base are frames, except primitive types like strings and numbers. There are four different types of frames (Noy, Fergerson et al. 2000):

- **Classes**: concepts of any domain, constituting a hierarchic taxonomy. Each class (except the root class) has one or more *superclasses*, from which it inherits *template slots*, and each class can have *subclasses*, which inherit its template slots.
- **Slots**: properties of frames, can be attached to classes as *template slots* or to any frame as *own slots*. Slot frames are first class objects; they exist on their own. Slot properties are defined as facets.
- **Facets**: properties of slots. Facets can be customized to the attached class.
- **Instances**: individuals of a certain class (type). All instances are derived from one and only one class. They acquire their own slots from the class's template slots.

At the same time all frames are instances of a certain class. Classes are instances of class metaclasses, slots are instances of a slot metaclass, facets are instances of a facet metaclass. They acquire own slots from the metaclass to describe their own properties. Even the metaclasses are instances of the root metaclass.

Each frame carrying own slots can acquire concrete values of its properties. A slot value is determined by a *frame/own slot* pair.

Protégé is written in Java and is provided as an *Open Source* project, so its source code is public and free. However, it is not necessary to change the original source code to create extensions. Thanks to the very flexible plug-in concept of Protégé own extensions can easily be integrated into the Protégé user interface.

1.3 Dataflow: Reading and Writing Data in a Knowledge Base

In order to enable the reading and writing access of external Java classes to frame/slot values the knowledge base system must provide an input/output interface. Fortunately the Protégé system supports user made extensions on three different levels. Since Protégé is written in Java the extensions must be developed in the same language.

Where is the most favourable point for external Java methods to be integrated in the dataflow of Protégé? As mentioned above, object attributes can be accessed by reading and writing own slot values of a certain frame. On the frame level the methods *frame.getOwnSlotValue(slot)* (read) and *frame.setOwnSlotValue(slot, value)* (write) could theoretically be used. In fact, it is not feasible to

create a new subclass of *Frame* to introduce the desired behaviour because this would not affect other subclasses of *Frame* (classes, slots, instances).

But there is another way to change input/output methods of frames: they don't have direct access to the frame database, but they call homonymous methods of the class *KnowledgeBase*. Subclassing of *KnowledgeBase* establishes the possibility of overriding the basic input/output methods.

The overriding read method first calls the external Java class method. The executable now can newly calculate the frame/slot value, before the inherited method accesses the database (see also fig. 1):

```
public Collection getOwnSlotValues (Frame frame, Slot slot) {
   callExternalFunction (frame, slot);                  //call external Java class (function)
   return super.getOwnSlotValues(frame, slot);          //call the inherited method
} // (simplified Java source code)
```

If an external Java class is used to restrict slot values (constraint), the incoming new values of the write method can be checked and blocked if they violate constraint rules. In this case the inherited write method will be avoided (see also fig. 2):

```
public void setOwnSlotValues (Frame frame, Slot slot, Collection values) {
   if (callExternalFunction (frame, slot, values)) {     //call external Java class (constraint)
      super.setOwnSlotValues(frame, slot, values);       //call the inherited method
      checkIfSlotIsInputSlot (frame, slot);              //check if slot is input slot of a function
   }
} // (simplified Java source code)
```

This version of the *callExternalFunction* method will solely return "false" if there is an external constraint and if this constraint detects rule violations. In all other cases the values will be stored in the inherited way.

The slot whose value has been changed may serve as an *input slot* (see below) of a function. In this case the function has to be re-calculated, since the change of the *input slot* might cause a change in the function result. The method *checkIfSlotIsInputSlot* checks if the slot is an *input slot* of any functions. If so, these function slots will be recalculated.

Now the infrastructure for accessing the frame database and controlling the dataflow is set up. It is positioned deep inside the *Protégé* architecture, but nevertheless it can be implemented as a *backend plug-in* rather than by modifying the program code. *Backend plug-ins* are usually developed to introduce other file formats.

2 The Java – Frame Interface

The backend plug-in has implemented modified input/output methods in order to allow external Java methods to control the dataflow. Now we have to talk about the way of handling external Java classes on the user level.

2.1 Activities

Potential activities can be seen as active properties of actors (objects), in contrast to passive attributes. As properties are commonly introduced as slots in frame logic, you can treat activities as a special kind of slot in a frame based ontology. A similar approach has been proposed for dynamic scheduling (Barták 2000).

So we suggest to link executables (which represent activities) to "*carrier slots*", although other solutions might be technically feasible. *Carrier slots* can be attached to all classes which feature the represented ability.

2.2 Carrier Slots

The above-mentioned Java method *callExternalFunction* first checks if the calling slot is a *carrier slot* (that is: it carries a pointer to a Java class instance). Solely if this is true, the external Java class method can be executed. If not, the slot will show usual behaviour.

The name of the executable is treated as a special attribute (slot). It is attached to the *carrier slot*, so the Java class will be reloaded when the ontology is opened next time.

Carrier slots provide the infrastructure to embed an external Java method. They deliver all necessary information the Java class needs to work in a proper way. If you want to relate any frame to an executable, a certain *carrier slot* referring to the external Java class has to be attached to the frame.

This construction causes that the entire dataflow between a frame and an executable must pass a *carrier slot* as an intermediate frame. That is, from the point of view of all frames (except *carrier slots*), functions, activities, and constraints appear as *internal frames* (slots) instead of external Java classes; therefore they can be handled like "normal" slots.

2.3 Parameters: Input Slots and Output Slots

When the executable is called by a frame, it receives some basic information about which frame was calling and about the associated *carrier slot*. This information is necessary but not sufficient if the method's task requires more detailed information, unless the external Java class uses explicit slot names to access the required slot values.

To solve this problem a convention of passing data is required: the suggested solution is oriented towards function calls in procedural programming languages. Functions as well as methods of object oriented languages use call parameters to pass the data to be calculated. To map this model on ontologies *input slots* can be assigned to carry the required data.

The result of a function call usually is returned by the function itself. Following this convention the function result can be represented by the carrier slot itself. That is, the result will be stored in the

carrier slot of the certain frame, and also will be returned by the *getOwnSlotValue* method. In this case the type restrictions of the *carrier slot* must match the expected function result.

If a carrier slot represents a more complex process instead of a function (which returns only one single value), the external method can cause changes of several slot values, maybe even values of different frames. To configure storing of output values the certain slots can be assigned as *output slots*.

Both, the input slots list and output slots list, appear as *multiple instance slots* (allowed class: the *slot metaclass*) which are attached to the *carrier slot*.

The issue which parameter slots are needed depends completely on the job the executable has to deal with. The Java class expects the parameters in a certain order. The advantage of this convention is (as mentioned above): Slot names are not relevant for the procedure, so the executables can be used in different ways and by different ontology classes.

2.4 The Basic Java Function Class

One of the main benefits of object oriented programming is the inheritance of methods. So we can create a basic abstract Java class which provides a framework of methods to administrate the import and storage of frame/slot values. Furthermore it provides the abstract core method *execute()*, which has to be overridden by any concrete function class. This method doesn't need to care about acquiring and storing data, it can focus on its specific calculations.

If the execution has terminated successfully the output values will be stored in the output slots of the current frame.

2.5 Operation types and Java Interfaces

How can a mismatch of Java classes be avoided? All usable Java classes must carry a basic signature. That is, they must implement a *basic interface*, which declares all methods that are used to provide an infrastructure for calling external executables. The *basic Java function class* does so, and all subclasses of it inherit the basic interface.

As indicated above executables can be employed in different ways and for different tasks. To distinguish Java classes by their task they can implement additional interfaces. So one can avoid misuse of them if they were linked to a not-suitable *carrier slot*.

Three fundamental *operation types* can be distinguished:

- *functions* calculate the values of certain slots on a reading access to the *carrier slot* value.
- *constraints* check new slot values on the writing access to the *carrier slot*; they can block the access if the values violate certain rules.
- *actions* are the most general type; must be triggered (usually by the user), then can do anything, e.g. change some slot values, create or delete instances etc..

2.5.1 Functions

Java classes implementing the *Function* interface are executed within the *getOwnSlotValue* method (see fig. 1). The executable can update the carrier slot value before it is read from the database.

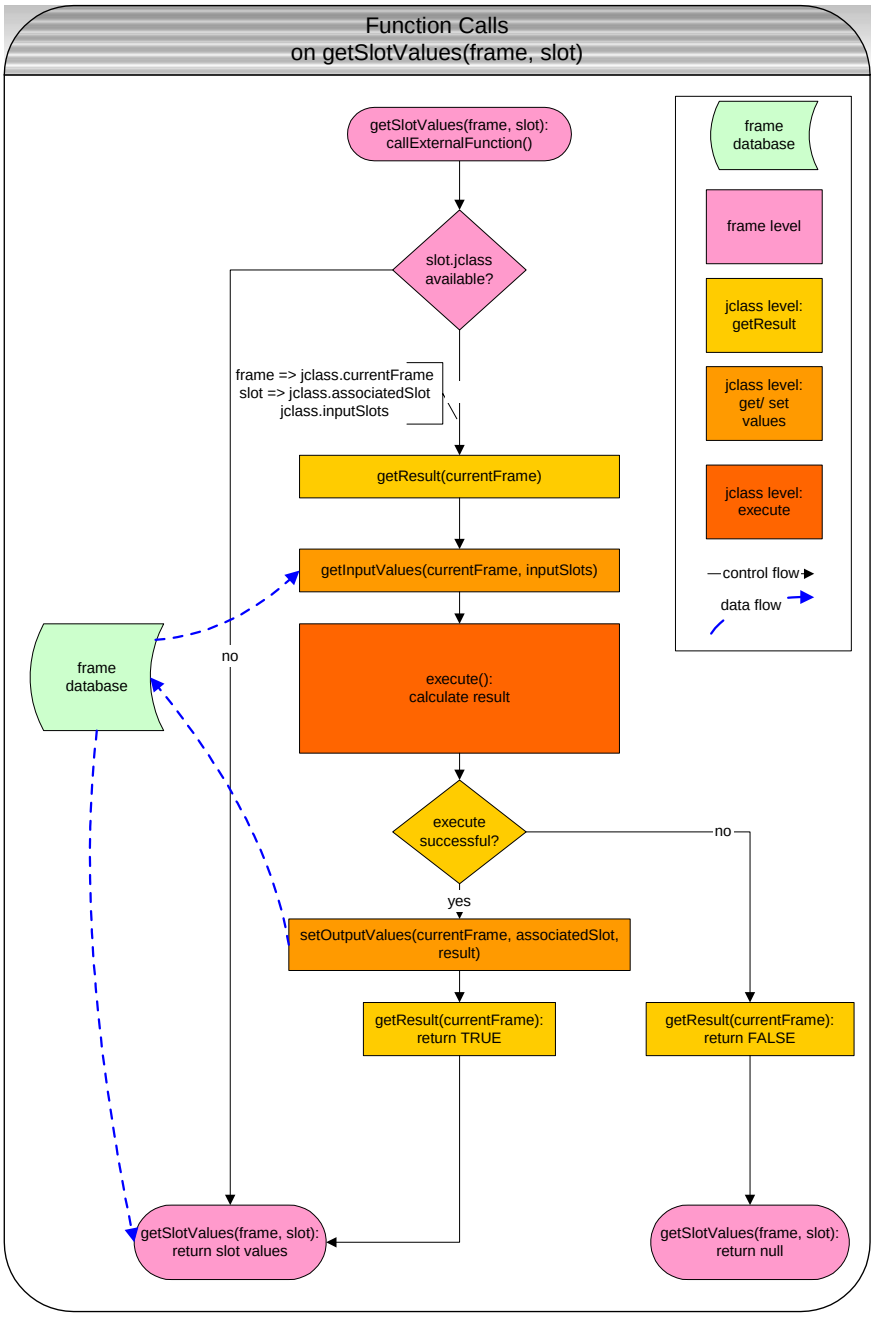
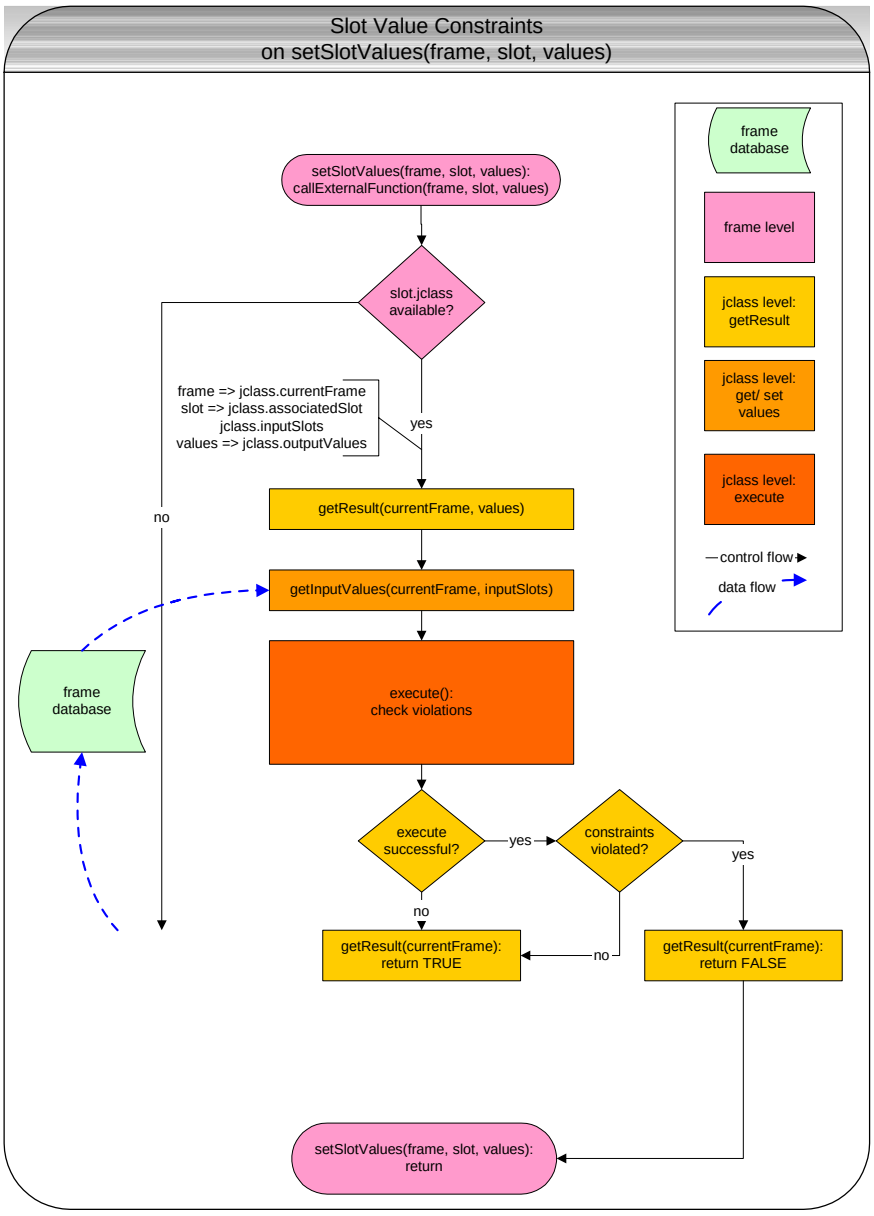


Fig. 1: Flow chart of a function call on the frame level.

Function classes can be used for all kinds of functions. That is, function routines return only one single value to be stored in the *carrier slot* itself, for example arithmetic and statistic functions, string functions, system functions like current date and time, and so on. The result can be a list of values; in this case the carrier slot must allow multiple values of a certain type.

A special task of function slots can be the visualization of *reified relations* or the string concatenation of other slot values.

In general function slots can be used for numeric and textual *evaluation*, *backward chaining* (proving goals) and *inferences* (reasoning, logical deduction) (Sowa 2000). They deliver the function result immediately and automatically when input data has been changed.



2). If the constraints are violated the new value will be rejected, otherwise it will be stored. In case of a violation the system should provide an error message with information about the reason why the value was refused.

If the values have been accepted the *checkIfSlotIsInputSlot* method checks if the slot in question serves as an *input slot* of any function. In this case these functions resp. *carrier slots* must be re-calculated because an input value has been changed.

Unlike the PAL constraints (another extension of *Protégé*) executable constraints act immediately on user input. PAL (= *Protégé Axiom Language*) delivers a list of instances which violate certain rules when the user starts a PAL constraint check. On the other hand PAL constraints can be created and edited within the *Protégé* user interface in a logical language (Crubézy 2002).

2.5.3 Actions

Actions and processes (composed actions) may not be executed on reading frame/slot values; in contrast to functions they have to be triggered by the user (e.g. by a mouse click on a button) or by other actions (see fig. 3). Therefore they actually do not need a *carrier slot* because there is no automatic execution and no function result to be stored in the *carrier slot*. Nevertheless *action carrier slots* are needed to represent actions and to attach them to certain classes.

If the action changes some frame/slot values the concerned slots must be passed to the executable as *output slots*.

Action slots can evaluate logical rules (encoded in the *execute* method) to manipulate data in the sense of *forward chaining* (Sowa 2000).

Action slots provide a wide range of activities to be started by “pressing a button”. Also, they are very loosely connected to the carrier slot – in fact, they don’t need it. So it is not quite sure if general actions should be represented by a slot. In most of the use cases it may be useful to attach an action slot to a certain class, but the knowledge engineer should use his liberty carefully in order to prevent a misuse of the ontology as an arbitrary kind of construction kit. An action should be a real property of the class it is attached to, and not execute anything regardless of the ontology.

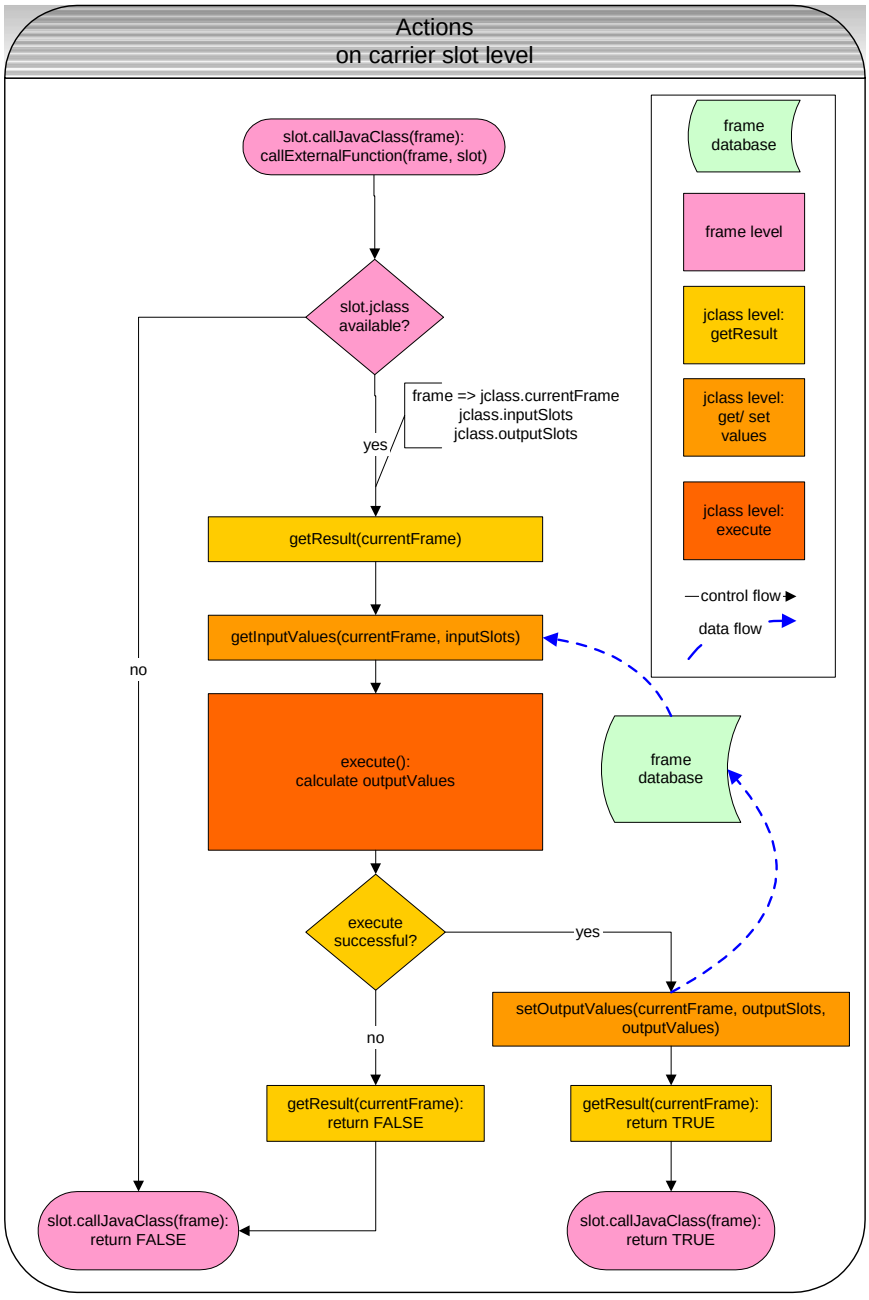


Fig. 3: Flow chart of an action execution on the carrier slot level.

3 Discussion and Future Work

This paper shows how a frame-based system like *Protégé-2000* can be enabled to execute external functions on the frame level. In this way conceptual knowledge can be combined with procedural and executable knowledge.

When functions represent activities or methods of objects (resp. frames) they can be linked to *carrier slots* which have to be attached to the frames. Concerning frames, all functions, activities, and constraints are represented by internal frames (*carrier slots*) instead of external Java classes; therefore they can be handled like properties.

External functions are called by a list of parameters – in a similar manner as *procedures* and *methods* in programming languages. Certain slots of the frame in question serve as input or output parameters. Before the core method of the executable (*execute*) can calculate the function result, the input values have to be derived from the frame and the input slots. After successfully finishing the execute method the output values have to be stored in the frame's output slots.

3.1 System requirements

Of course the solution described in this paper is a special one for the combination Protégé – Java classes, but it seems to be portable to other constellations under certain conditions.

It was very helpful to use the same programming language (Java) in which the ontology editor was developed. But that is not a necessary precondition. Probably each object-oriented programming language is suitable, maybe even procedural-only languages.

The knowledge base system must essentially provide an *application programming interface* (API) which allows at least reading and writing access on frame/slot values. This would yet enable implementation of *action* interfaces (see fig. 3) executing triggered by user command. They even can work without a *carrier slot*, and applications of this operation type can be provided by any conventional Protégé plug-ins.

To establish the *function* and *constraint* interface (see fig. 1 rep. 2) is a little bit more complicated: the executable has to be notified when a frame/slot value is going to be read or written. In case of the *constraint* interface the executable should be able to reject the new slot value, at least it should be able to prompt an error message.

Must the knowledge system be a frame-based one? Generally this doesn't seem to be a necessary condition. In any case the action-carrying entities must be identified to be linked to executables. Nevertheless, due to the lack of experience I actually cannot answer this question, but I guess that the power of the API is a more important issue than the kind of knowledge model used by the system.

3.2 Future work: Signatures and metadata

In this paper one issue has not been addressed at all yet: how can an executable notify the user (e.g. the knowledge engineer) which parameters are required? That is, there is some information needed about the *meaning* of the required parameters, the *order*, and the *data types* as well.

As long as the user and the developer of the executables are the same person, all may work fine. Problems appear at least if a user wants to embed an executable without having further information.

So it turns out that there is something needed like *metadata*, describing the interface. External functions should provide *signatures* to indicate meaning, number, order and type of input and

output parameters resp. slots. The signatures could be shown in the input form of the carrier slot where the user configures the Java - Frame interface. So the user can check immediately if the input and output slots match the signature.

The next step: a generally adaptable description language for functions and services to make executables interoperable for all users should be applied to the Java – Frame interface. The user should be able to easily recognize the ability of the function and whether it is usable for the desired task or not. Maybe this information can automatically be acquired by a *carrier slot* to show properties concerning the function in the input form. The approach of (Gaio, Lopes et al. 2003) could show a suitable way of describing interfaces for executables by predicate definitions in the markup language DAML-S.

4 References

An implementation of the described features (documentation and download) is available at http://ifgi.uni-muenster.de/~borcherr/protege/functioncalls.html and as well at the Protégé “Plugins” page (http://protege.stanford.edu/plugins.html) under “Inference & Reasoning”.

Protégé-2000 Programming Development Kit. http://protege.stanford.edu/doc/pdk/index.html, http://protege.stanford.edu/.

Barták, R. (2000). Slot Models for Schedulers Enhanced by Planning Capabilities. Nineteenth Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG), Milton Keynes, UK, December 2000, Milton Keynes, UK.

Crubézy, M. (2002). The Protégé Axiom Language and Toolset ("PAL"). http://protege.stanford.edu/plugins/paltabs/pal-documentation/, http://protege.stanford.edu/.

Freire, J. and L. Botelho (2002). Executing explicitly represented protocols. Workshop "Challenges in Open Agent Systems" of the 1st International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2002).

Gaio, S., A. Lopes, et al. (2003). From DAML-S to Executable Code. Agentcities: Challenges in Open Agent Environments. B. Burg, J. Dale, T. Fininet al, Springer-Verlag**:** 25-31.

Mota, L., L. Botelho, et al. (2003). O3F: an Object Oriented Ontology Framework. 2nd International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2003).

Noy, N. F., R. W. Fergerson, et al. (2000). The knowledge model of Protégé-2000: combining interoperability and flexibility. 2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France.

Raubal, M. and W. Kuhn (forthcoming 2004). "Ontology-Based Task Simulation." http://musil.uni-muenster.de/documents/ontologies_simulation.pdf.

Sowa, J. F. (2000). Knowledge representation: logical, philosophical, and computational foundations. Pacific Grove, CA, Brooks/Cole.