Knowledge-based system development with scripting technology – a recommender system example

Dietmar Jannach
Department of Computer Science
Dortmund University of Technology, Germany
e-mail: dietmar.jannach@udo.edu

Abstract

The core functionality of many knowledge-based systems is built with the help of special-purpose software components and programming environments such as rule engines or Prolog interpreters. Other parts of the application – like the Web interface – are however built with "standard" software development technology like Java which means that not only the corresponding interfaces and data exchange mechanisms between the components have to be developed, but also that the software developers have to work with different technologies or even programming paradigms.
In this paper we show how recent "scripting" extensions in a programming language like Java can be exploited to develop highly flexible and extensible knowledge-intensive applications. The different advantages of such an approach are discussed based on the experiences gained from developing a scripting-based software library for building knowledge-based recommender applications.

1. Introduction

The possible advantages of a knowledge-based software development approach are well known. The domain knowledge can be separated from the reasoning knowledge while at the same time optimized off-the-shelf reasoning engines can be used to automate the inferencing or problem solving process.

In many real-world or commercial applications, however, only a part of the "intelligent" system will be developed with the help of special programming environments such as LISP or Prolog or by use of a rule-engine like Jess[1]. The web interface or the database layer will most probably be developed with the help of "standard" technology such as Java, Servlets, or Java Server Pages. This mix of technologies brings additional complexity to the software development process since programmers not only have to implement the different interfaces and data exchange channels but – more importantly – are faced with different programming paradigms.

In the field of programming languages, we can observe signs of a revival of "scripting" languages in recent years. Despite their disadvantages with respect to type-safety, compile-time problem detection and run-time performance, languages like PHP, Python and recently Ruby became popular in particular for web application development as they – according for instance to [13] – promise to be advantageous with respect to flexibility and developer productivity. From our point of view, the aspects of flexibility and in particular extensibility that these interpreted languages provide also make them interesting for the development of knowledge-intensive systems.

In this paper, we report and discuss our experiences gained from developing the light-weight JPFINDER library for building knowledge-based recommender systems . The library is fully written in the Java programming language and exploits the language's recent "scripting" support to achieve the required levels of extensibility and compactness of the knowledge bases. Overall, the work shall thus exemplify one of the options of embedding knowledge-based system development into industrial software development processes and environments.

2. Application domain & system architecture

In the field of recommender systems, the following main approaches can be distinguished: Classical *community-based recommender* systems base their proposals on item ratings, user similarities, and collaborative filtering techniques, see [1] for a recent overview. *Content-* or *knowledge-based* systems on the other hand operate on the basis of further pieces of information, which can for in-

[1] http://www.jessrules.com/jess/index.shtml

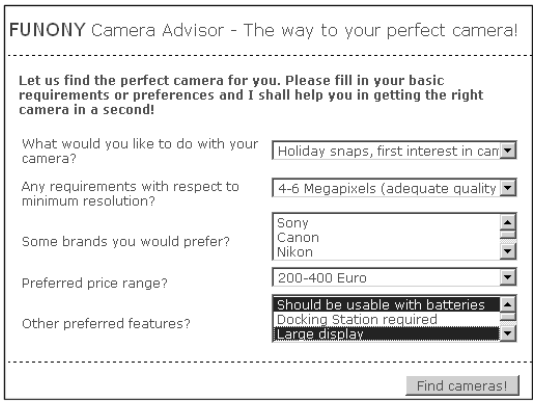

Figure 1. Preference elicitation

Figure 2. Recommendation & explanation

stance be knowledge about the items to be recommended and/or knowledge about directly or indirectly acquired preferences of the users. Typical techniques used in the latter type of systems are for instance filter-based matching, similarity-based retrieval or utility-based ranking [4].

Screenshots of a typical interactive recommender application in the sense of [5] are shown in Figure 1 and Figure 2. The user is first guided through a series of possibly personalized questions in order to determine her specific needs (Figure 1). At the end of this dialog, the system comes up with a recommendation and – due to its knowledge-based nature — is also capable of explaining the reasons for this particular proposal, retrieve other similar catalog items, or let the user revise or specify additional requirements.

It can be easily observed that such applications are *knowledge-intensive*, meaning that various pieces of domain-specific information have to be encoded in the background. Examples for such knowledge chunks are for instance the *matching rules* that determine which items suit some given requirements, utility functions for ranking the items, or personalization rules to adapt the user interface and navigation path according to the current user profile.

In [5], CWADVISOR, an integrated environment for the development of such knowledge-based and highly interactive recommender systems is presented. The CWADVISOR system is designed as a classical and to some extent heavy-weight expert system. It relies on the explicit representation of domain knowledge, provides a fully-fledged graphical knowledge acquisition component, a relational database for persisting the knowledge as well as proprietary languages for modeling filtering or personalization rules.

The JPFINDER library discussed in this paper implements many of the features of CWADVISOR and was designed to be a light-weight alternative to the comprehensive CWADVISOR expert system. In particular it should also take advantage of recent scripting features of the Java programming language as to reduce the development efforts that are required for rule processing while at the same time extensibility and flexibility should be retained.

In the work presented herein, we will particularly focus on how the *rule knowledge* is represented and processed, as the development of software systems that are governed by *business rules* are in wide-spread industrial use also in other application domains. In the CWADVISOR system, a proprietary rule language is used. The statements are either parsed, translated and compiled to Java code [8] or directly executed with the help of a proprietary rule interpreter [5]. In either case, a comparably complex parser and compiler component as well as a graphical tool for modeling the rules (including for instance auto-completion and correction features) are required, whereas JPFINDER relies on scripting technology for that purpose.

The overall architecture of the JPFINDER system is outlined in Figure 3. At the core, the *Recommender Engine* interacts with the different users of the system and correspondingly maintains *Recommender Session* objects that contain the current user's profile. The *Engine* has a defined Application Programming Interface to manipulate the domain knowledge which can also be used for loading the definitions from a persistent data store at system startup. Several pluggable *reasoning modules* are also part of the library and implement specific functionalities such as filter-based matching, utility calculation or techniques for user interface personalization.

3. Recommendation technique implementation

In the following, we will discuss the logic of JPFINDER and some of its modules in more detail and in particular focus on the scripting-based implementation of the functionalities.

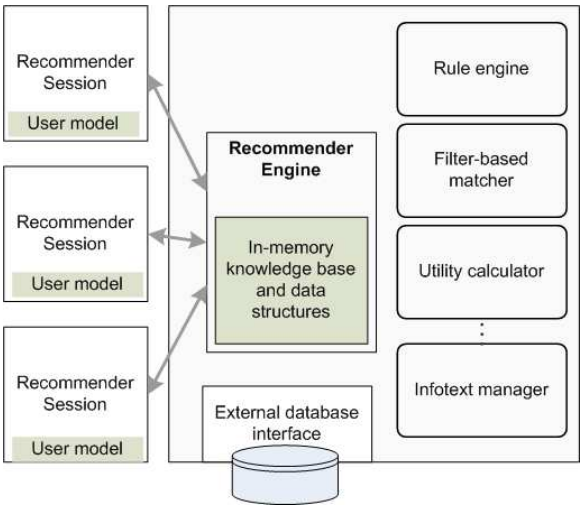

Figure 3. Architecture overview.

3.1. The user & product model

In JPFINDER, every piece of information about the user and the preferences used for generating recommendations are contained in the *user model*. In order not to limit the generality of the approach, the user model is thus generally defined to be a finite set of variables *UV* (user variables), each of them with a defined data type. Variables can be either string-valued or numeric; of both types, multi-value instantiations are possible, see for instance the choice of preferred features in Figure 1. Note again that the values for variables in *UV* do not necessarily have to be directly acquired through questioning but can also be derived "internally", e.g., based on scoring schemes or other value derivation rules. The set of specific (input) values for one individual user shall be denoted as *IV*. In fact, also complex user modelling techniques – that for instance maintain a history of user interactions – can be employed. The implementation of such profiling or learning techniques is however beyond the scope of JPFINDER.

Similarly, the *product model* defines the characteristics of the items to be recommended. The product characteristics are described by a set of variables *PV* (product variables) with defined data types. Beside regular product features, the product model can also be used to describe "external" product characteristics such as vendor reliability or current stock availability in a given shop.

Each individual recommendable item is thus described by a set of key-value pairs; the set of all items forms the product is denoted as product catalog *PC*.

In the example, *UV*, *PV*, *PC* and some user inputs IV_1 could be as follows, using Java notation for data types.

$$UV = \{pref\_price : Double, \\ pref\_features : String[], ..\}$$

$$PV = \{price : Double, resolution : Double, ..\}$$
$$PC = \{\{price : 400.00, resolution : 3, ...\}, \\ \{price : 300.00, resolution : 2, ...\}\}$$
$$IV_1 = \{pref\_price = 100, pref\_features = ..\}$$

3.2. Filter-based matching

Model. With *filter-based matching* we mean a technique in which customer preferences are directly or indirectly mapped to constraints on product properties. Such filter-based approaches are also commonly used in similarity-based systems to pre-filter the set of products to be compared [11].

In JPFINDER, the mapping from user preferences to desired product characteristics can be expressed in the form of "if-then-style" rules, i.e., a filter rule $FR < AC, FC >$ consists of an *activation condition AC* and a *filter constraint FC*. We use $FR.AC(IV)$ to refer to the evaluation of the activation condition given input values IV and $FR.FC$ to refer to the filter constraint definition.

A typical filter rule in the domain could be: *"**If** the user prefers the lower price range **then** recommend items with a price lower than 200 Euro."* Formally, filter rules are interpreted as follows. Let AC be an expression over variables in UV, FC an expression over the variable set $UV \cup PV$. IV are the customer input values and PC is the product catalog given in the form described above.

Given these inputs, the "catalog query" Q is thus the conjunction of FC-expressions of those filter rules for which AC evaluates to *true*, i.e.

$$Q \equiv \bigwedge(f.FC)|f \in FR \wedge f.AC(IV) = true$$

If we interpret the product catalog PC as a relational database table, filtering the set of recommended products RP corresponds to performing the database query σ_Q on PC.

Scripting-based implementation The implementation of filter-based matching in JPFINDER relies on a recent algorithm [9] which also supports fast query relaxation for failing queries and which is based on compact in-memory data structures and partial in-advance query evaluation.

Note that although there have been several efforts to defining a common knowledge interchange format (see, e.g., KIF[2]), no common representation mechanism for rules or constraints is yet broadly established. In more recent efforts, the general constraint language ESSENCE [7] has been proposed and SWRL[3] has been submitted to the W3C as a rule language in the Semantic Web. Still, besides problems of syntactic complexity (SWRL) or limited expressiveness (ESSENCE), special purpose parsers, compilers, or interpreters have to be employed to support these languages.

[2] http://logic.stanford.edu/kif/kif.html
[3] http://www.w3.org/Submission/SWRL/

```
<xml>
  <FilterRule>
    <if condition="c_pref_price_range = 'medium'"/>
    <then constraint="p_price > 200 && p_price < 400"/>
    <explanation text="I could obey your price preferences
                       (medium price range)"/>
    <excuse      text="Note that due to your other requirements I could
                       not obey your price preferences"/>
    <priority="3"/>
  </FilterRule>
  <FilterRule>
      <if condition="..."/>
                  ...
</xml>
```

Figure 4. XML representation of filter rules.

As a knowledge representation mechanism for the filter constraints, JPFINDER therefore relies on JavaScript, which is the default scripting language supported by the recent Java 6 SE release. As described above, the filter rules are written in a simple "if-then"-style, which in our experience (see, e.g., [5]) is easy to comprehend also for domain experts who are not experienced in programming or specific knowledge representation techniques. The activation condition and the filter constraint are formulated as JavaScript expressions over the variables of the user model.

In Figure 4, an example of a filter rule including relaxation priorities and explanation texts in the sense of [9] is shown. At run time, the variables of the user model – which may have been acquired via an ordinary Java Server page or from a more elaborate user modelling component – are forwarded to the scripting engine which then evaluates the activation conditions of the rules. Those expressions that evaluate to true are then used to filter the suitable catalogue items. Note that The XML representation in Figure 4 is optional as the library provides an appropriate Java-based application programming interface for registering the filter rules.

Within the expressions statements, all JavaScript language constructs can be used and arbitrarily complex calculations are thus possible. The actual values of the variables of the user- and product model are automatically put into the scope of the scripting engine. With respect to extensibility aspects, the chosen scripting approach also allows us to define custom JavaScript functions that can be seamlessly used within expressions. Consider, for instance, that a filter constraint should be written that is activated whenever the user has chosen a particular value from a *set of options* like *"**if** the customer preferences (among others) contain 'usb' **then** ..."*.

To that purpose, the knowledge engineer can write a standard JavaScript program that tests for set membership:

```
function isContainedIn(value,um_var) {..}
```

Custom functions like this can then be registered to the recommender engine at runtime and used within the filter constraint. Internally, the script code are automatically compiled into Java code for performance reasons, which is a standard feature of Java SE. The only task of the recommender engine is to put the current values of the session into the execution scope when the function is called.

3.3. Utility- and similarity-based retrieval

Retrieving items based on their expected utility to the user or based on the similarity with (individual features of) another item are two other techniques used in knowledge-based recommendation, see [2] or [3].

At the core of *utility-based approaches*, a *utility function* is used whose value either depends on specific product features alone or which also takes the user's preferences into account. A typical evaluation scheme for determining such personalized utility values can for instance be based on Multi Attribute Utility Theory (MAUT) [8, 14].

Within JPFINDER, such arbitrarily complex utility functions can be defined in a similar way like the custom extensions mentioned above, i.e., with the help of JavaScript functions.

```
<UtilityFunction type="dynamic">
      function myUtility() {
            var utility_total = 0;
            // Utility dimension mobility
            var utiltiy_mobility = 0;
            if (p_weight < 300)
                  utiltiy_mobility += 3;
            if (p_batteries = 'yes')
                  utiltiy_mobility += 2;
            ...
            // Weighting based on user prefrences
            if (pref_mobility == 'high')
                  ...
            return utility_total;
      }
</UtilityFunction>
```

Figure 5. Fragment of utility function.

A fragment of a possible utility function that both evaluates product features and user preferences is sketched in Figure 5. At run time, JPFINDER applies the function on

each catalog item separately. The resulting utility values can then be used to sort the items in the recommendation list accordingly. JPFINDER supports two variants of utility functions, *static* and *dynamic* ones. If a utility function is marked to be static then no user model variables must be used in the function. This differentiation is mainly introduced for performance purposes. If no user model variables are used, the values of the utility function will be the same for all customers, which in turn means that the corresponding values can be pre-computed when the library is initialized. "Dynamic" evaluation functions have to be evaluated in the context of the requirements of a specific user like in the MAUT approach.

In the same way, custom functions can be developed to implement *similarity-based retrieval* recommendation techniques. In contrast to utility functions, the return value of a similarity function is not an individual utility value but rather a normalized numerical value that describes the relative similarity between two items. Based on this mechanism, JPFINDER thus supports the implementation of recommender systems like the Wasabi personal shopper [3] or of critiquing approaches [10], in which the user can ask the system to retrieve items that are similar to a given one with respect to some features.

3.4. Rule-based inferencing

In some application domains, additional forms of inferencing (on user model variables) are required. This could be for instance the "internal" derivation of further variable values based on business rules or the determination and personalization of dialog pages in an interactive preference elicitation process [5].

JPFINDER supports the implementation of business rules or additional personalization logic through generic extension patterns that allow the developer to write code fragments, which are executed when certain conditions are fulfilled.

```
<BusinessRule>
  <If>
     c_pref_investment_duration > '10 years'
  </If>
  <Then>
     if (c_pref_question_1 == 'yes')
        c_derived_score += 3;
     if (...)
        …
  </Then>
</BusinessRule>
```

Figure 6. Fragment of business rule.

Figure 6 shows a fragment of a possible business rule in JPFINDER: Depending on some user input the value of *c_derived_score*, which corresponds to an internal variable of the user model, is determined. Note that the consequent of the rule can contain arbitrary code, which means it can also be used to perform mathematical calculations for, e.g., repayment rates commonly used in the financial services domain [6].

Personalized Text Fragments represent another extension pattern of JPFINDER. With the help of rules of this type, the recommender system can select personalized variants of predefined text fragments, which can for instance be used to adapt the graphical user interface according to the knowledge level of the current user. Finally, the pattern of *User Model Expressions* allows the engineer to define arbitrary expressions over user model variables. Based on the evaluation of these expressions for the current user, the dialog flow of the Web interface or other personalization features of the application can be designed in a flexible way.

The implementation of the rule execution engine in the current version of JPFINDER is a rather simple one. For the case of business rules, the engine for instance simply evaluates the rules until no more changes in the user variables can be observed. More elaborate techniques like rule-chaining or more expressive types of rules are planned for future versions.

4. Implementation aspects

Run-time performance is in general one of the key issues for interpreted scripting languages. Thus, particular attention has been paid to these aspects both in the design as well as in the implementation of JPFINDER's algorithms. What became obvious quite soon is that the "immediate" execution of text-based scripts in Mozilla's Rhino engine[4], which is the standard implementation in Java 6 SE, is not applicable for realistic problem sizes.

With respect to algorithms, let us exemplarily discuss the filter-based matching and relaxation problem from section 3.2. In particular the search for optimal "relaxations" of a failing query can be a costly operation as theoretically all combinations of subqueries have to be evaluated. In contrast to previous algorithms used for this task [11, 12], JPFINDER thus implements a novel technique [9] which is based on the in-advance evaluation of the filters and the usage of compact in-memory data structures. It was shown in [9] that this way the number of required "database queries" for determining the optimal relaxation can be limited to the number of given subqueries at the cost of slightly increased memory requirements.

Beside the usage of such recent algorithms, JPFINDER also relies on run-time "compilation" of all scripting code. When the library is initialized with the external knowledge base or additional knowledge pieces

[4] http://www.mozilla.org/rhino/

are added via the API, all JavaScript expressions and functions are automatically translated into Java byte-code. This functionality is implemented based on the built-in Java's `ClassCompiler` and dynamic class loading. Consequently, when scripting code has to be evaluated in a recommendation session, the script interpreter is actually not needed anymore as everything is already available in the form of Java byte code. Still, no manual recompilation is required when the knowledge base changes, as the needed byte code can be generated at run time.

As an example for performance numbers, consider the following rough running time numbers measured on a standard desktop computer (Intel P4, 3.2 GHz, 1 GB RAM). A recommender knowledge base for digital cameras may comprise around 400 products and 30 filtering rules, which in our experience is a realistic size. Let us assume that 15 of the filter rules are "active" in a current session and the best relaxation comprises 12 rules, i.e., three filters have to be relaxed. The running time for such a problem setting (without dynamic filter evaluation as described in [9]) is around only 100ms. Even if we double the problem size (800 products and 60 rules), the response time is still less than half a second, which is appropriate for interactive recommender sessions. Another number can be given for the dynamic construction of new filter conditions, see the "a bit cheaper models" - functionality in Figure 2. Evaluating for instance a single five-atom query with a smaller price for a catalog of 800 cameras requires less than 20ms. The overall memory requirement for both examples is below 15 megabytes, including all the product data which is kept in memory.

5. Conclusions

Based on an example from the domain of knowledge-based recommender systems, the paper has demonstrated how scripting technology can be used to simplify the development of knowledge-intensive software applications.

Compared with complex and heavy-weight expert systems that incorporate specialized programming environments or rule-processing engines, the advantage of the presented approach in particular lies in the fact that the software and knowledge engineer is not confronted with different programming paradigms and languages. In addition, the use of a consistent set of technologies simplifies the integration of such intelligent reasoning modules into standard Web development toolkits and industrial software development processes.

The preliminary evaluation of the presented library, which is based on real-world scenarios and experiences gained from previous projects [5], indicates that (1) many of the functionalities of more complex frameworks can be implemented with less code (as no special parsers, compilers, or interpreters are required), and that (2) performance issues that commonly arise in scripted languages can be successfully addressed with the help of runtime compilation.

References

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.

[2] D. Bridge. Product recommendation systems: A new direction. In R. Weber and C. Wangenheim, editors, *Workshop Programme at 4^{th} Intl. Conference on Case-Based Reasoning*, pages 79–86, 2001.

[3] R. Burke. The wasabi personal shopper: a case-based recommender system. In *Proceedings of the 11th National Conference on Innovative Applications of Artificial Intelligence, AAAI'99*, pages 844–849, Menlo Park, CA, USA, 1999.

[4] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2005.

[5] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. An integrated environment for the development of knowledge-based recommender applications. *International Journal of Electronic Commerce*, 11(2):11–34, Winter 2006-7 2007.

[6] A. Felfernig and A. Kiener. Knowledge-based interactive selling of financial services using FSAdvisor. In *17th Innovative Applications of Artificial Intelligence Conference (IAAI)*, pages 1475–1482. AAAI Press, 2005.

[7] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernández, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *International Joint Conference on Artificial Intelligence - IJCAI*, pages 80–87, Hyderabad, India, 2007.

[8] D. Jannach. Advisor suite - a knowledge-based sales advisory system. In L. S. Lopez de Mantaras, editor, 16^{th} *European Conference on Artificial Intelligence (PAIS)*, pages 720–724. IOS Press, 2004.

[9] D. Jannach. Finding preferred query relaxations in content-based recommenders. In *IEEE Intelligent Systems Conference*, pages 355–360, Westminster, UK, 2006. IEEE Press.

[10] L. McGinty and B. Smyth. Adaptive selection: An analysis of critiquing and preference-based feedback in conversational recommender systems. *International Journal of Electronic Commerce*, 11(2):35–57, 2006.

[11] D. McSherry. Retrieval failure and recovery in recommender systems. *Artificial Intelligence Review*, 24(3/4):319–338, 2005.

[12] N. Mirzadeh, F. Ricci, and M. Bansal. Supporting user query relaxation in a recommender system. In 5^{th} *International Conference on E-Commerce and Web Technologies (EC-Web)*, pages 31–40, Zaragoza, Spain, 2004. Springer.

[13] B. A. Tate. *From Java to Ruby*. Pragmatic Bookshelf, 1st edition, 2006.

[14] D. von Winterfeldt and W. Edwards. *Decision Analysis and Behavioral Research*. Cambridge University Press, Cambridge, UK, 1986.