

The Matching Systems Engine and Rules Engines

Simon Field, Technology Director, Matching Systems 2007 Ltd.
spf@matchingsystems.com

Rules Engines, such as products from Ilog and Blaise, are marketed as generic solutions to a wide range of problems that demand the application of business rules. The Matching Systems Engine is a product that addresses dynamic matching, configuration and selection problems across a range of business domains. There would therefore appear to be an overlap in the solution spaces that can be addressed by these tools. Are these products competing in the same space or are they complementary?

1. The Solution Space

A clue to the answer is found in the description of the Matching Systems Engine's solution space. It is limited specifically to tasks involving the matching and selection of solutions from a set of possibilities, with the added ability to involve dynamic configuration as part of the matching process. For the sake of simplicity, we shall hereafter refer to this problem space as *matching problems*. Whilst *matching problems* can involve the use of business rules, they are not the only kind of problem that demands their use. The range of problems addressed by Rules Engines is therefore wider and, as will be shown, the use of a Rules Engine to address *matching problems* demands more development and maintenance effort than would be the case if the same problem was addressed with the Matching Systems Engine. This results in a higher risk to the project, compared with using the Matching Systems Engine, or a combination of the Matching Systems Engine with a Rules Engine (in just the same way that other problems ideally suited to Rules Engines can also be addressed with programming languages such as Java, but again with higher costs and risk).

2. Requirements for matching solutions

Researchers at IBM's Zurich Research Laboratory have for many years explored the nature of *matching problems* and solutions, with a particular focus on the added complexity demanded by the matching and selection of complex products and services (as compared with commodity goods). The following characteristics were identified as essential to the successful application of technology to *matching problems*:

Symmetry of selection: The selection process must be two-sided where both sides have to describe themselves in terms of what they offer and are therefore able to specify what they demand of the other party.

Power of description and power of requirement expression: powerful languages are needed to describe complex services and similarly, powerful languages are needed to describe complex compatibility criteria, i.e. requirements on the description of the other party.

Ability to specify configurable services: enabling the service description to be completed dynamically at the time of matching. This is needed for several reasons:

- It facilitates an up-to-date description of the service where the cost, availability or quality of service may be subject to variations. Such variations can for example be dependent on the requirements or circumstances of the customer.
- It provides a way to specify a range of services without having to enumerate all the options associated with them.
- It provides a way to configure the service according to the needs and properties of the both parties. This facilitates personalisation of the service.

Facilitate the structured intervention of third parties: the ability to ask for third-party evaluation of a service or a customer or the integral participation of third parties where combined/bundled services are concerned, for example.

Ability to distribute the matching process: allowing more commercially sensitive information to remain within the service-providing organisations, rather than requiring all descriptions to be shared centrally.

Ability to stage the matching process: allowing both parties to stage the revelation of information to each other, facilitating the sensitive handling of personal or commercially sensitive information, while also providing a more natural, phased approach to the search process through what is likely to be a complex information space.

3. Requirements and business rules

Each of these characteristics is enshrined in the Matching Systems Engine. It may be noted that only two of the six, *Power of description and power of requirement expression* and *Ability to specify configurable services* relate in some way to the possible use of business rules. It is therefore reasonable to suppose that a Rules Engine with its business rule language might be a good starting point for addressing *matching problems*. However, the need for the descriptive and requirement languages to be rich and expressive imposes some constraints on the nature of the business rules language that can be adopted.

There is clearly a trade-off between richness of expression and ease of use. At one end of the scale is a full programming language, with reusable methods, loop constructs, inheritance and the like, and at the other end is a simple point and click form-based language, where the structure is fixed, but values and names of variables can be entered or selected from drop-down lists. The findings of the Zurich Research team were that many real-world problems have complex information structures that require navigation from the requirements language. For example, a requirement specification may need to sum a total from a repeating structure, and handle some exceptional circumstances, as part of the business rules determining the suitability of a particular solution. This suggests that for *matching problems*, the trade-off should favour richness of expression, sacrificing some ease of use.

Many business rule languages used by Rules Engines adopt a simple *If...then...* structure that is easy to use, but requires a flat list of named attributes and is unable to cope with a rich information structure, lacking the necessary control constructs with which to navigate repeating structures and the like.

It is an attractive idea to use the same language to configure services as is used to specify requirements. This is the case with the Matching Systems Engine, and would equally be so if a solution was implemented with a single Rules Engine. However, it is highly desirable that the two functions are clearly separated. The Matching Systems Engine achieves this because the matching and configuration processes are not implemented with rules, but are built into the core Engine. The location and use of rules that relate to the selection of an advertised service is distinct from the location and use of any configuration rules, which are attached to descriptive properties. This clean separation makes for a simpler description of service advertisements, better runtime performance, and results in speedier implementation, reduced risk of error, and lower maintenance costs. Use of a Rules Engine to implement these two types of rule may demand additional rules or coding to manage the distinct selection and configuration functions.

Even if the language used by a Rules Engine does have the required richness of expression and division by function, it only satisfies two of the six requirements described in Section 2. above. Any solution developed with that rules language is therefore likely to require further development effort to implement the remaining four characteristics. This will involve either the development of extensions to the Rules Engine, or the development of capabilities that satisfy these requirements using the rules language itself. This latter course is often quite feasible, as the rules language is itself a form of programming language, whose use can go beyond the expression of business rules. However, such a course of action carries with it the risk of mixing together true business logic with rules that manage the matching and configuration processes. This destroys one of the key advantages of using business rules – the separation of business logic from process definition and enactment, and causes management and maintenance difficulties similar to those created by implementing such solutions entirely using conventional programming languages.

4. Requirements and Matching Systems Engine

The Matching Systems Engine was developed to address the entire list of requirements described in Section 2. As stated at the beginning of this document, it aims to solve *matching problems* rather than a broader range of unspecified “business rule” problems. Each of the six characteristics is built into the core Engine, allowing the service provider to focus solely on providing a description of the entities to be configured and matched, and leaving the process of matching, configuring and selecting, to the underlying capabilities of the Engine.

This clean separation of search requests and the descriptions of the entities to be matched from the underlying matching, configuration and selection process enables *matching problems* to be solved very quickly, reducing development costs, and minimising ongoing maintenance costs and project risks. Changes to the content involve amendments to, and submission of, simple xml documents.

The matching process applied by the Matching Systems Engine can be extended, but this is done using the java programming language, with simple plug-ins that conform to specified java interfaces, and not by (ab)using the language with which

advertisements are described. This avoids the danger of mixing the business rules with the matching process, and helps to maintain the critical separation of the two.

Given the need for the rule language to be “rich and powerful”, the default language used for requirement expressions is a powerful subset of java (in essence, the body of a java method). This allows users to include loops and other control structures, local variables, and even call methods in externally provided java classes. It also delivers excellent run-time performance, as the resulting service advertisement is converted by the Engine into a compiled java class. As indicated above, providing such a richly expressive language comes at the cost of sacrificing some ease of use. This default language is too rich and complex to be supported with a simple “point and click” graphical user interface. The Matching Systems Control Panel, for example, gives the user simple text panels in which to enter the expressions that make up selection or configuration rules.

It can be that, for some business domains, a different trade-off between richness of expression and ease of use is desirable. This can be achieved in two ways:

- Use of a graphical user interface to limit the user to a given logic structure, which may, for example, take the form of *if...then...* rules. These are, in effect, a small subset of the default capability, and so the user interface can generate the appropriate underlying java code for use by the Matching Systems Engine, thus protecting the user from the complexity (and flexibility) of the java language.
- Incorporation of a java-based extension that implements a higher-level language. This might be a rules-based language, or a script language. JRules from Ilog and Python are examples of each, both of which can easily be added to the Matching Systems Engine to provide alternative rule and dynamic property languages.

5. Conclusion

Rules Engines have been designed as general-purpose environments well suited to implementing business rules. Business rules play a part in addressing *matching problems*, but on their own lack a number of essential characteristics that otherwise need to be implemented as extensions.

The Matching Systems Engine has been designed and developed specifically to address *matching problems*. The clean separation of the matching, configuration and selection process from the description of solutions, search requests and the specification of requirements allows for very rapid development and deployment of matching solutions, and ease of maintenance, thereby minimising both the cost of development, and the risk of project failure. The Matching Systems Engine’s use of a rich and expressive requirements language can, if desired, be replaced or supplemented with the embedded use of a script language such as Python, or a Rules Engine such as JRules.

The Matching Systems Engine is not a general-purpose programming environment, but it can address a very wide range of *matching problems*. These are characterised by the need to select one or more solution from among a potentially large set of possibilities, which may need to be configured dynamically to suit the circumstances of individual situations. To illustrate the range of problems that fit the *matching problems* pattern, here are some examples taken from the insurance industry:

- Customer needs analysis – selecting the appropriate solutions and advice for a given set of customer circumstances
- Dynamic product composition – selecting and configuring product components for inclusion in a product offering
- Underwriting and pricing – determining and matching the right combination of premium rate and terms and conditions for a given customer risk profile
- Fraud detection – comparing the circumstances of an insurance claim with those of known fraudulent claims
- Process selection – choosing from among a set of possible ways of managing a newly reported claim, according to the circumstances of the claim, and the profile of the customer concerned

If a business problem fits this pattern of *matching problems*, use of the Matching Systems Engine will yield substantial benefits in reduced development and maintenance costs, and reduced project risk.