

Business Users and Program Variability: Bridging the Gap

Isabelle Rouvellou, Lou Degenaro, Judah Diament, Achille Fokoue, Sam Weber

{ rouvellou, degenaro, djudah, achille, samweber } @ us.ibm.com
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Abstract. In order to make software components more flexible and reusable it is desirable to provide business users with facilities to assemble and control them, but without first being converted into programmers. We present our fully-functional prototype middleware system where variability is externalized so that core applications need not be altered for anticipated changes. Application behavior modification is fast and easy, suitable for a quickly changing e-commerce world.

1 Introduction

Domain experts see business processes as "themes and variations". When describing a process they first describe its broad sweep or theme, then come back to fill in the details as a sequences of variations on the theme. In most cases, these variations capture business practices or policies that change more often than the rest of a business process. Changes can come from inside (e.g., a new discount strategy) or be mandated from outside, typically a regulatory agency (e.g., federal tax rate computation). Traditionally, these policies have been buried in application code. Externalizing them from the rest of the application has many advantages. It makes the business policies explicit, increasing both application understanding and consistency of business practices. It also allows the enterprise to lower the cost of application maintenance.

This paper describes our fully-implemented infrastructure and framework, Fusion, that allows applications to be structured and developed such that the core behavior of the business process is built-in while the variations are managed and applied externally by business users (or non-programmers – we use these terms interchangeably). We will present the overall Fusion architecture and show how it addresses three fundamental goals we strived for: empowering the business users to define and manage policies, easing the integration with and migration from existing applications, and finally minimizing performance impact.

We will discuss related work before describing our system in detail.

2 Related Work

The system described in this paper builds on our previous work "*Extending Business Objects with Business Rules*" [7], inspired by its deficiencies plus new requirements.

Previously, we provided programmers facilities to systematically externalize variability for management by business users - via trigger points, externally parameterized code fragments, and a management tool. Our approach was effective (a successful product resulted), but many capabilities from design-to-production were limiting or non-existent, especially for business users.

Using terminology introduced in “*On the Notion of Variability in Software Product Lines*” [9], our improved middleware system provides “open variation” points following the “multiple parallel variants” pattern. We describe how variability is implemented, but leave for the future discussion of some management issues.

2.1 Expressing Externalized Variable Logic

How should business users express externalized variable logic? OCL [4] is recommended by the Object Management Group for defining constraints in models. Although easy to read and write, OCL is a pure specification language - expressions are side-effect free, and implementation issues are not addressed. This wasn't suitable for us, since we need to express computation including data modifications.

SBA [11] allows non-programmers to describe and manipulate information using tables, business forms, and reports. Automation is accomplished by giving “examples” to the system on how to manipulate information. The techniques used in SBA do not extend easily to less data centric applications and the advocated “query by example” paradigm, essentially requires business users to learn a programming language.

2.2 Managing Externalized Variability

How should externalized variability be managed, design-to-production? The Variation Point Model [10] represents variation within a model. Complementary to our work, the technique presented could be used for identifying, within core applications, points of variability and information available at them (i.e., context). Our work further suggests that the variability need not be programmed by a programmer, but can be authored by a business user.

“*Support for Business-Driven Evolution with Coordination Technologies*” [2] recognizes that software development techniques, such as object oriented ones, do not effectively address software evolution. A coordination contract discipline is introduced that helps in this domain, but it targets software developers, not business users.

JAsCo[3] is a set of aspect-oriented techniques for programmers to connect business rule implementations with business rules-driven applications. These techniques could automate the insertion of variability points in applications (see section 3.3).

2.3 Externalized Variability Systems

A number of commercial enterprises offer rule-based software [14-17] as a way to enable non-programmers to specify application logic. Most of these rule-based systems execute using inferencing engines. Because rules are declarative, they are considered

“natural” to business users. Business users can indeed parameterize templates, but more general logic authoring requires a skilled IT person with knowledge of inferencing concepts (e.g., working memory, implicit iteration ...). As pointed out in our previous paper, [8], inferencing is appropriate to solve a number of problems, but our experience is that most externalized variable logic doesn’t require inferencing. Meeting with potential users, we learned that many were unwilling or unable to employ inferencing-based systems because the results are not 100% reproducible.

“*Evaluating Expert-Authored Rules for Military Reasoning*” [6] enables subject matter experts to manage knowledge directly without formal logic training. The focus is on writing sensible rules by non-logicians for an inference engine, while our concentration is on enabling authoring of simple, autonomous, and memory-less statements.

“*Naked Objects: a technique for designing more expressive systems*” [5] unveils a toolkit to expose object methods using a noun-verb style. It uses reflection to compose a lone user interface view. It allows for just one language based upon the subject Java object collection. It requires that participating Java objects implement the Naked Objects interface raising concern that the business objects themselves become “bloated” in order to be all things to all people. In keeping with one of our main goals, easing the integration with and migration from existing applications, our work does not place any requirements on participating objects, such as the implementation of a particular interface; we allow for many views of objects; we facilitate construction of customizable languages permitting renaming, hiding, and composing for each customized vocabulary constructed.

3 Fusion

The Fusion distributed middleware system is comprised of two types of components, some used at runtime and others used for development (see Fig. 1). There are three basic runtime components: the rule-enabled application (e.g., a Web Application), the Connector Registry, and the Runtime Engine. Each of these is briefly described below; more details are given later.

Applications using the Fusion middleware system are manufactured by programmers in the usual way; with the exception that rule-enablement occurs through the embedding of one or more “Points of Variability” (PoVs), where appropriate. PoV embedding is either a manual or an automatic patterned process. Development of the application and placement of PoVs in the core application still requires programming skills, but this is usually done just once and not changed very often.

The Connector Registry provides for a level of indirection between rule-enabled applications and the Runtime Engine. This allows rules to be shared between applications without creating dependencies between them.

The Runtime Engine executes the rules on the production server.

There are five basic development components: Vocabulary Authoring, Logic Authoring, Connector Authoring, Code Generation, and Artifact Management. There is also a Deployment component which takes the development artifacts and deploys them to runtime entities. Again, each of these components is briefly described below; more details are given later.

The Vocabulary Authoring component provides the means to define terms that can later be used in the construction of sentences using the Logic Authoring component. Vocabulary terms are comprised of two basic parts: properties and mappings. Properties are used mainly to display information to business users, while mappings are used to produce executable forms. For example, the alias property for a particular vocabulary term might be “the customer’s name” while the invocation mapping might be “Customer.getName()”.

The Logic Authoring component enables business users to express business logic in a mistake-proof environment, syntactically speaking. Whether constructing compound sentences, performing calculations, or making assignments, business users manipulate familiar terms, such as “the customer’s name”, while the Logic Authoring component assures that the underlying parameter and return types match accordingly.

The Connector Authoring component provides for indirect, type-safe, efficient linkage between rule-enabled applications and sets of business rules. It produces PoVs for inclusion with (and in order to) rule-enabled applications, and a runtime business rule set selection mechanism and transformation service.

The Code Generation component transforms the business logic authored by business users expressed in vocabulary terms familiar to them into an executable form. This component processes the outputs of the Logic Authoring and Vocabulary Authoring components to produce Java class files.

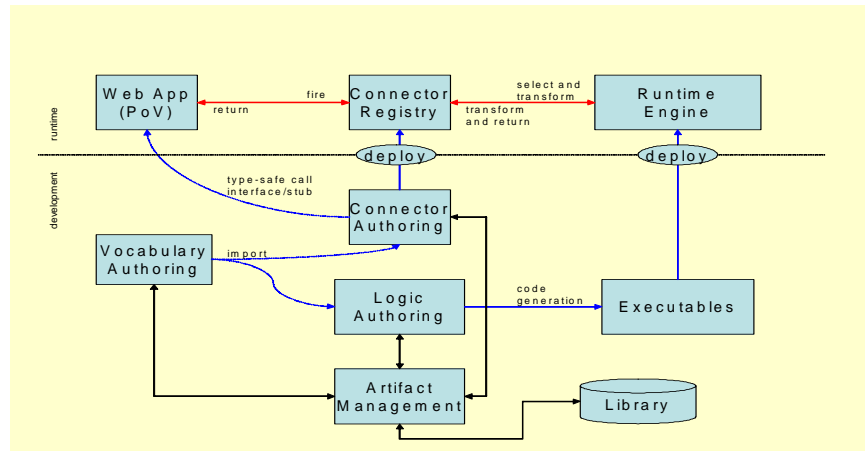


Fig. 1. Fusion externalized business logic development and runtime components

The Artifact Management component and the Library form a repository of Fusion produced artifacts in eXtensible Markup Language Metadata Interchange (XMI) format. By formalizing and separating the data in a structured way, extensions and transformations are readily possible.

The development and deployment aspects of the Fusion system were built as a collection of plug-ins to the Eclipse [12] platform plus a browser-based tool. We utilize the Eclipse Modeling Framework (EMF) [13] to manage the meta-model of the development-time artifacts as well as their persistent instances.

3.1 Vocabulary

One of the fundamental problems that we faced was the inherent mismatch between the data structures used by the application programmers and those usable by business people. Programmer developed Java code is not understandable by non-programmers. Therefore, we decided to offer the business users a more appropriate data view.

The Vocabulary Authoring component presents to business users a simple class model, without inheritance. The model consists of a set of entities corresponding to business concepts. An entity can have attributes and relationships to other entities as well as operations that can be performed. The names of entities and members are arbitrary, unlike Java: “The next business day” is a perfectly legal operation name. Operation arguments can be infix, so that “Cancel orders pending more than ___ days”, can be the name of an operation, where “___” represents the operation’s argument. For convenience, users can organize entities into a hierarchy to ease development of systems with large numbers of classes.

Naturally, the members and the relationships presented to business users have to be implemented and related to the objects that the application will pass to the Fusion Runtime Engine. A possible design choice (and one taken by a number of other business rule vendors) is to define Java classes that correspond directly to the model presented to business users, requiring application programmers to use those classes only. We felt that this was unacceptable for two reasons:

1. Since the vocabulary model is unlikely suitable for the application’s purposes, the application programmer is forced to translate the application’s data to and from the objects constructed for business users. This translation process adds additional burden onto the application programmer, and is likely error-prone.
2. Business rules are subject to frequent change. Whenever a change requires data and/or behavior that are new, the business model will have to be changed. If the application program has to consume the objects constructed from this model, the application program will be subjected to these changes in turn. This would defeat the purpose of externalization.

Our solution was to support translation from application program objects to vocabulary business entities, so that both the business user and the application program can use objects suitable for their purposes. This is made more difficult by our resolve to make defining this translation process as easy as possible, and to enable the translation to be updated when the business model changes.

The Vocabulary Authoring component maintains three data structures:

- The vocabulary model, which defines the business user’s model: what business entities, members and relationships are available.
- A model of a subset of the application programs. Specifically, we keep track of information about all the application classes and members that are either going to be passed to or from the Fusion Runtime Engine, or which are necessary to implement the vocabulary model.
- The “mapping information” which states how each vocabulary business entity, member and relationship is implemented.

We allow vocabulary entities to differ a great deal from the application classes by which they are ultimately implemented. Although each instance of a vocabulary entity must correspond to an application object, each member and each relationship just has

to be implemented by a type-correct series of operations performed upon application objects. For example, the vocabulary might include “Customer” and “Order” entities, in which an operation on Order provides “Does this order’s value exceed the customer’s limit?”. This might be implemented by the code “(Client.getMaximumOrderLimit(this.getClient()) > this.getTotalPrice())”, where “Client” is an application class that maps roughly to the vocabulary entity “Customer”.

In detail, the mapping information keeps for each vocabulary member and each relationship an abstract syntax tree (AST) describing its implementation. This AST is rich enough to include all the computational power of Java expressions. The AST refers only to information found in the application class model, so that if the application classes are modified, we can determine the impact of these changes upon the mapping information. Likewise, when the vocabulary model is changed, we can determine what mapping information is affected.

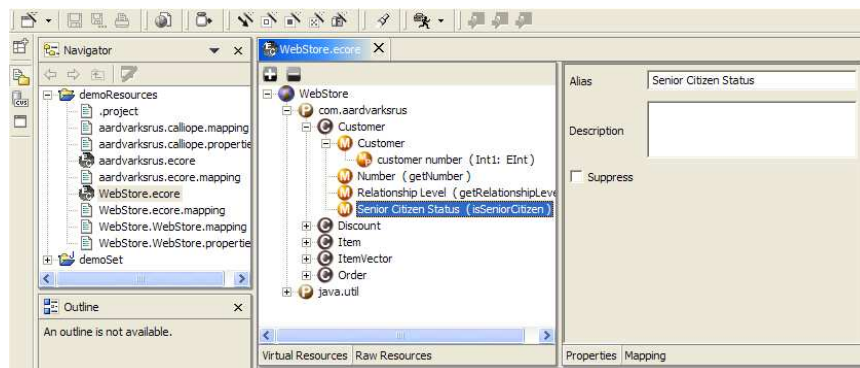


Fig. 2. Fusion Eclipse-based vocabulary management tool

We provide tools for constructing vocabulary models (and their associated mapping information). One tool (shown in Fig. 2) inspects Jar files containing Java class files (no Java source is necessary), and constructs a vocabulary model corresponding precisely to these classes. The user is then given the ability to:

- Omit classes which business users shouldn’t use
- Omit members which aren’t appropriate
- Rename entities and members to be more appropriate
- Construct new vocabulary members which don’t correspond to any application members
- Change the implementation of any vocabulary members
- Hide a method on class A returning class B as the mapping information for a relationship between corresponding entities A and B

To support use-cases where users desire to build a vocabulary model independently of existing artifacts (either because they are not suitable for business users or because application building is proceeding from scratch), we have another tool that takes a UML model and constructs a corresponding vocabulary model. The mapping information can then be added with our mapping tool. Thus, Fusion tools support both a bottom-up and a top-down approach to defining business models.

3.2 Logic Authoring

We provide two tools for authoring rule sets, rules, and templates: one for technically capable business users and a second for those less sophisticated. The former is an Eclipse-based tool that allows authors to manage all aspects of rule sets, rules, and templates, including creation, deletion, and significant modification. The latter is a web-based tool that constrains authors to limited changes, such as enabling/disabling a rule, or completing a template by choosing an option from a pull-down menu. We discuss below the Eclipse-based tool.

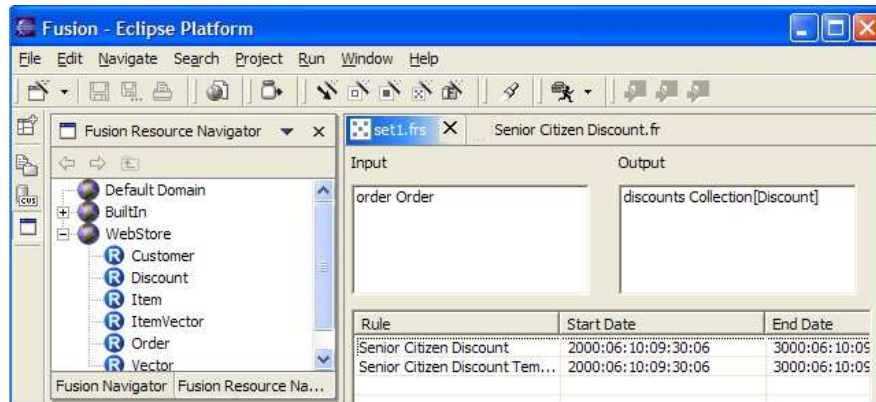


Fig. 3. Fusion Eclipse-based rule set authoring

Rules and templates are organized into rule sets. Before rules and templates can be authored, a rule set must first be created. The rule set author creates the rule set and gives it a name. From the available vocabulary (previously defined using the vocabulary authoring tool described in Section 3.1), the rule set author selects the inputs to and outputs from the named rule set. The inputs to a rule set are collectively known as the input group, and similarly the outputs from a rule set are collectively known as the output group (see Fig. 3).

Once a rule set is created, individual rules can be added, updated, or deleted. To add a rule to a rule set, first the rule author supplies a rule name. Next, the rule author selects terms from the scoped vocabulary (i.e., specific object instances) to create an if-then-else statement. The scoped vocabulary available for authoring a rule is determined by the input and output groups of the containing rule set. It includes the terms in the input and output groups, plus terms that are navigable from them through relationships, as well as some built-in vocabulary (such as Date, Time, String, Number, equals, less than, etc.). Terms can be selected and dragged into an if-then-else form by the rule author (see Fig. 4). For example, the rule set author might define “order” in the input group and “discounts” in the output group. The rule author might then define a rule using the scoped input and output as follows:

```
Senior Citizen Discount rule: If customer is a senior citizen and
order total is greater than or equal to 200 then add 10% senior
citizen discount.
```

In the above example, the name of the rule is “Senior Citizen Discount rule”. The rule author drags and drops vocabulary terms and operators onto the if-then-else form. The vocabulary terms used in the example are “customer is a senior citizen”, “order total”, and “discount”. The built-in vocabulary terms used in the example are “and”, “is greater than or equal to”, and “add”.

The rule author is prevented from making syntactic mistakes. For instance, when using the “is greater than or equal to” vocabulary term the underlying numeric types must match. In the above example, vocabulary term “order total” is an integer that can be compared to another integer vocabulary term or expression. Any attempt to drag-and-drop an incompatible type will fail with a pop-up message.

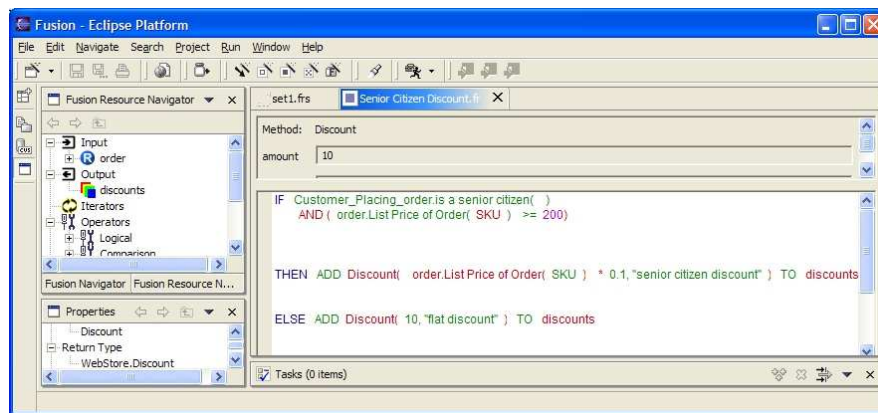


Fig. 4. Fusion Eclipse-based rule authoring

One controversial issue which surfaced was the perceived requirement by some that the authored rules always be syntactically correct, even during construction of complex statements. This often forced the rule author to author multi-conjunctive clauses in a particular order, which we ourselves and our users found irritating.

A template is comprised of a rule that has portions designated as substitutable. Subsequently, a new rule can be created by filling in the substitutable portions. The rule template author selects an existing rule for templatizing, gives the template a name, and selects those portions subject to substitution. Each substitutable portion is assigned a variable name by the rule template author.

Senior Citizen Discount Template rule: If customer is a senior citizen and order total is greater than or equal to <amount> then add 10% senior citizen discount.

In the above example, the name of the rule template is “Senior Citizen Discount Template”. The rule template author selected from the if-then-else form the term “4” and assigned the substitution variable name “amount”. New rules can be defined based upon rule templates by less skilled business users employing the browser-based tool.

Once rule sets, comprised of rules and rule templates, have been completed, they are deployed to a test or production system using Fusion deployment tools. Once deployed to the runtime, the rule sets are available for use by rule-enabled applications.

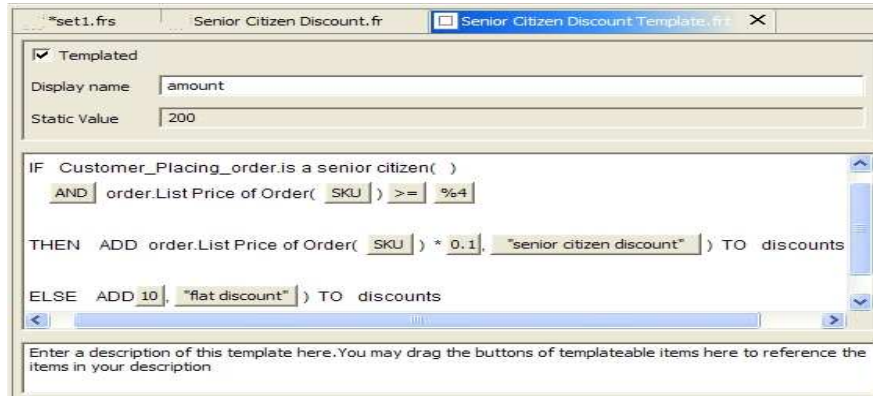


Fig. 5. Fusion Eclipse-based template authoring

3.3 Application Connectivity

The Vocabulary and Logic Authoring components empower skilled and less skilled business users with the ability to author business logic outside an application program proper. The functionality offered to business users is limited - not nearly as powerful as programming languages such as C++ or Java. Programmers are still called upon to author the core application. Connecting the core application with the externalized business logic is the challenge met by the Application Connectivity component.

This component serves a number of purposes. First, it provides indirection between the application request for service and the externalized business logic (rule sets) authored by business users. Second, it insures handshaking type safety between the core application and the externalized business logic. Asking the programmer to construct and pass arrays of objects, as was the case in our prior work [7], is prone to error. Third, it promotes definition and externalization of selection criteria (e.g., in the insurance industry, the business logic for a particular business purpose could vary by date, U.S. State and product). At runtime, the Fusion selector framework employs the external information to bind a particular business logic to a particular application request. The remainder of this section explains how these three objectives are met.

The application connectivity component produces two types of artifacts: Points of Variability (PoVs) and a Connector Registry comprised of Logical Operations (LOs). The former are generated code stubs, deployed with the core application, containing methods invoked whenever externalized business logic is employed. A PoV includes a name, selection criteria signature, an evaluation signature, and a connected LOP name. The latter matches incoming PoV requests with desired rule sets based upon selection criteria. A development-time tool insures that the evaluation signature of the PoV matches or can be transformed into that of the associated LOP.

At development-time, the Connector Registry component is configured by providing the list of available LOs. An individual LOP is a contract (or signature), similar in purpose to Java interfaces. It is comprised of a name, input expected and output

produced (signature), a description (purpose), a collection of eligible rule sets based upon selection criteria (e.g., start-end dates, state), and a selection context signature (type of information that will be available at runtime to select the appropriate rule set). For example, a rule-enabled application may request that the “determine eligibility” logical operation be employed. The selection algorithm employed by the LOp for “determine eligibility” might be: link to rule set 1 on even numbered days; else to rule set 2 on odd numbered days. The input and output types are guaranteed to match.

At runtime, the Connector Registry listens for Business Logic Request (BLR) events fired by PoVs. A BLR event encapsulates information about the PoV, the selection context which specifies information used to select the most relevant rule set, and input parameters needed for the evaluation of the selected rule set. The Connector Registry processes a BLR event in three steps: match, evaluate and return. First, its selection framework finds the rule set whose selection criteria match the selection context of the BLR event. Second, the selected rule set is evaluated using the BLR event input parameters to produce desired output (data transformations are performed if required). Finally, the evaluation result is returned to the PoV, which returns it to the application.

In the future, this last step could be performed by firing a Response Event which would encapsulate the initial BLR event and the evaluation result. It would then be processed by the PoV that would be listening for such events. Other components (e.g., Business Activity Monitoring components) could also listen for these events. Furthermore, this communication model would permit Fusion runtime to easily work in conjunction with an event based reactive rule system such as AMIT [1].

Rule set selection is dynamically performed at runtime by the selection framework. Although selection logic is potentially arbitrary, we designed a default table driven algorithm which, we believe, covers most useful cases. The criteria definition corresponds to the structure of the table used by the selector, for example “select logic by date, state and purpose.” When connecting logic to applications, the business user specifies a particular instance of that criteria (01/01/2000, 06/14/2002, NY, insurance premium computation). At runtime the selector algorithm matches instance values encapsulated in Business Logic Request Events against the selector tables. Our design handles an arbitrary number of dimensions restricted to be of simple type (numeric, enumeration, string) and possibly constrained (integer between 1 and 100).

Making the selection framework a first class citizen in the architecture achieves two objectives. First, it makes rule management easier and more scalable by introducing a highly structured partitioning of rules. For example, it enables users to work with a particular subset meaningful to an application (e.g., NY rules). In a typical scenario, a user updates the rules to match regulatory changes in a specific state. Presenting the rules separately by state in the tools helps the user focus on the rules for that particular state, without the distraction of rules for other states. Given additional restrictions on the criteria dimensions (e.g., lower and upper bound for numeric), we are able to verify common business consistency requirements such as complete coverage (e.g., check that a premium computation logic exists for every state). Second, it enables better runtime performance by sub-setting the rules that are considered at runtime. In the insurance example, when a policy component is invoked for “NY”, the rule engine need only evaluate the “NY” rules, ignoring those for all other states. Rules are selected with a more efficient SQL lookup resulting in a performance boost compared with a design that lumps the rules for all 50 states together.

3.4 Deployment and Runtime

Runtime performance and correctness are critical. While during development it is common for there to be (temporary) inconsistencies between artifacts, the deployment process must detect and prevent inconsistent updates to the production server. We must also optimize rule set execution, to maintain an acceptable level of performance.

Our runtime engine associates each Logical Operation with a specific data flow between Nodes, where each Node is the executable representation of a rule set. Currently, the executable representation of a Fusion rule set is a Java class file. However, our runtime engine is general-purpose: it supports a plugin mechanism whereby other rule set representations can be executed. In fact, earlier in our research process we were using another executable representation for Fusion rule sets, and switching to Java only required creating a new plugin – no changes to the engine itself.

The deployment process takes place in two steps. First, the output of the development tools is transformed into a deployment file. In order to create it, we first check that all development artifacts are consistent: the implementations of our Logical Operations create values of the correct type, and so forth. Then, each Fusion rule set is translated into a Java class file. The output of this process is a single file containing all necessary information for deployment to a server.

In a second step, a deployment file is deployed to a server. Another set of consistency checks is done to make sure that the server state is what the deployment file expected. This prevents the system from crashing if the server state accidentally gets out of sync with the development system. Finally, the system state is updated in a transactional manner so ongoing operations always experience a consistent system.

4 Conclusions

Fusion addresses our three main objectives: usability, integration, and performance. We've shown the architecture of a prototype middleware that facilitates business user control of exposed aspects of applications in a systematic and manageable way without need for programmer assistance. We've shown development and runtime components of our middleware, describing the purpose and operational characteristics of each.

The Fusion system provides application connection componentry which makes integration with existing and future rule-driven applications fairly simple. It introduces minimal runtime overhead, thus having minimal negative runtime performance impact. Fusion does **not** introduce new runtime objects to support business user vocabularies which might adversely affect performance, but rather uses existing objects unmodified. Fusion does provide indirection for rule set selection through a connection registry which does have some minimal performance cost, but with the added advantage of increased flexibility. Ongoing investigation is addressing outstanding issues including: integrating variability concepts in the overall application lifecycle; extracting legacy rules from code; and rule management. Under the rule management umbrella, query capabilities are needed to answer business user questions such as: which rule sets are now in force; which rules produce discount results; which rules are concerned with senior citizens; and so forth. Additional meta-data and facilities are needed for

managing versions, dependencies, consistency, and, at a higher level, to understand compliance with business policies and the impact of proposed changes to name a few.

We postulate that middleware, embedded systems, and others will employ externalized logic techniques in order to create more reusable off-the-shelf components in the near future. Using such techniques allows for late binding of core general purpose applications with customizations appropriate to the surrounding environment. It also shortens the turn-around time between expressed desire by the application owners and the programmers normally required to make the appropriate adjustments.

References

1. Adi, Biger, Botzer, Etzion, and Sommer, *Context Awareness in AMIT*. Proceedings of the Fifth Annual International Workshop on Active Middle Services, June 25, 2003, Seattle, Washington, IEEE Computer Society.
2. Andrade, Fiadeiro, Gouveia, Koutsoukos, and Wermelinger, *Support for Business-Driven Evolution with Coordination Technologies*. Proceedings of the 4th International Workshop on Principals of Software Evolution, 2001, ACM Press, New York, NY.
3. Cibran, D'Hondt, Suvee, Vanderperren, and Jonckers, *JAsCo for Linking Business Rules to Object-Oriented Software*. Proceedings of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications (CSITeA '03), Rio De Janeiro, Brazil, June 2003.
4. Eriksson and Penker, *Business Modeling with UML*. Wiley & Sons, New York, NY, 2000.
5. Pawson and Matthews, *Naked Objects: a technique for designing more expressive systems*. ACM SIGPLAN Notices, Vol. 36, Issue 12 (Dec. 2001), ACM Press, New York, NY.
6. Pool, Russ, Schneider, Murray, Fitzgerald, Mehrotra, Schrag, Blythe, Kim, Chalupsky, and Pierluigi, *Evaluating Expert-Authored Rules for Military Reasoning*. Proceedings of the International Conference on Knowledge Capture, 2003, ACM Press, New York, NY.
7. Rouvellou, Degenaro, Rasmus, Ehnebuske, and McKee, *Extending Business Objects with Business Rules*. TOOLS Europe 2000, St. Malo, France.
8. Rouvellou, Degenaro, Chan, Rasmus, Grosf, Ehnebuske, and McKee, *Combining Different Business Rules Technologies: A Rationalization*. Workshop on Best Practices in Business Rule Design and Implementation, OOPSLA 2000, Minneapolis, Minnesota.
9. Van Gurp, Bosch, and Svahnberg, *On the Notion of Variability in Software Product Lines*. Working IEEE Conference on Software Architecture, WICSA '01 Proceedings, IEEE Computer Society, Washington, DC, 2001.
10. Webber and Goma, *Modeling Variability with the Variation Point Model*. Software Reuse: Methods, Techniques, and Tools 7th International Conference, ICSR-7 Proceedings, Springer, 2002.
11. Zloof and Jong, *The System for Business Automation (SBA): Programming Language*. Communications of the ACM, June 1977, Vol. 20, No. 6, ACM Press, New York, NY.
12. Eclipse 2004, www.eclipse.org
13. Eclipse EMF Project 2004, www.eclipse.org/emf
14. IBM Agent Building and Learning Environment 2003, www.alphaworks.ibm.com/tech/able
15. Corticon Home Page, 2004, www.corticon.com/home.html
16. Fair Isaacs Decision Tools, 2004, www.blazesoft.com
17. ILOG Home Page 2004, www.ilog.com