Twelve Rules for Business Rules

By: C. J. Date

Date: 5/1/2000

Introduction

This paper has to with what are commonly called *business rules*. Its purpose is to propose a set of rules about such rules—rules that, it is suggested, a "good rule engine" really ought to abide by. Such rules about rules might well be called *metarules*; they might equally well be described as *objectives*; however, this paper refers to them as **prescriptions**.

Disclosure

This paper was prepared under an agreement with Versata Inc., a company that has a business rule product to sell. However, it has categorically *not* been written in such a way as to "make Versata look good"; the various prescriptions it describes have been designed without any special reference to the current commercial scene in general or Versata's product in particular. In other words, the paper serves to document the writer's own opinion merely—the writer's opinion, that is, as to what business rule products ought to strive for in the future.

Assumptions

It is convenient to begin by stating some underlying assumptions and introducing some terminology:

- The purpose of any given piece of application software—an **application** for short—is to implement some **enterprise work item** (i.e., some piece of functionality that is relevant to the enterprise in question).

  *Note: The term **application** is used in this paper in a very loose kind of way; thus, a given application might be a simple subroutine (e.g., a function to calculate withholding), or a large collection of mutually interacting programs (e.g., a fully integrated corporate accounting system), or anything in between.*

- The enterprise work item in question is specified as a set of **definitions** (data definitions, access definitions, form definitions, and so forth).

- To the maximum extent logically possible, those definitions are **declarative**—i.e., nonprocedural—in nature. They are also **formal** (necessarily so, of course). In what follows, they are referred to as **business rules**, or just **rules** for short.

  *Note: The reason for the slight degree of hesitancy in the foregoing paragraph ("To the maximum extent logically possible") is that the rules in question might include certain **stimulus/response** rules, which do include an element of procedurality (see Prescription 3, later).*

- Business rules are **compilable**—i.e., mechanically convertible into executable code—and hence, loosely, **executable**. In other words, the set of rules that constitutes the declarative specification for a given application **is** the source code for that application, by definition (pun intended). Thus, the activities of (a) *specifying* or *defining* the application, and (b) *developing* or *building* it, are in fact one and the same.

  *Note: It follows from the foregoing that, so far as this paper is concerned, the terms **rules** and **business rules** are reserved to mean **rules that can be automated**. Other writers use the term more generally. For example, the final report of the GUIDE Business Rules Project [3] defines a business rule to be "a statement that defines or constrains some aspect of the business." By this definition, a statement to the effect that the last person to leave the premises must turn off the lights might qualify as a business rule—but not one that is very interesting from the point of view of business automation. To put the matter another way, not all business policies and protocols are capable of being automated, and this paper is concerned only with ones that are.*

- Finally, the software system that is responsible for compiling and overseeing the execution of such declaratively specified applications is called the **rule engine**.

A Note on Terminology

It is, unfortunately, undeniable that the term *rules* is not a very good one (certainly it is not very specific, nor very descriptive). And the term *business rules* is not much better; in particular, not all enterprises are businesses. As already indicated, this paper does make use of these terms; however, it does so primarily because other publications in the field do so too! Be that as it may, the really important point is that the technology under discussion is a **declarative** one. To be more specific, rules, whatever they might be called, are *declarative*, not *procedural*, in nature: Application software—possibly some system software too—is specified declaratively, and the resulting declarative specifications are directly compilable and executable.

Further Preliminary Remarks

It might be felt that prescriptions such as the ones to be discussed in this paper ought all to be independent of one another. After all, such independence is surely desirable for the same kinds of reasons that the prescriptions themselves demand certain kinds of independence in business rule systems. However, it turns out to be more convenient to state the prescriptions in a kind of layered fashion, with later ones building on earlier ones; thus, the various prescriptions are not, nor are

they claimed to be, fully independent of one another, despite any possible advantage that might follow from trying to make them so. What is more, some of the prescriptions overlap others (that is, some requirements are stated in more than one place, in more than one way).

Now, many readers will be aware that there are precedents for a paper of this nature. What is more, some of those precedents have become a little discredited over time, and the writer is therefore sensitive to the possibility of criticism (especially criticism along the lines of "oh no, not again"[1]). But business rule systems are becoming increasingly important, and it does therefore seem worthwhile—the foregoing comments notwithstanding—to attempt to provide some structure for the debates that will inevitably arise in the marketplace. Thus, what follows is offered as a kind of yardstick, or framework, that might conveniently be used to orient such debates; equally, it is proposed as a basis against which business rule systems might be "carefully analyzed, criticized, evaluated, and perhaps judged" (wording cribbed from reference [2]).

That said, however, there are a couple of important caveats that need to be spelled out immediately:

- The prescriptions that follow are emphatically **not** intended as a basis for any kind of "checklist" evaluation. To say it again, they are offered as a framework for structuring discussion and debate; they are definitely not meant as a basis for any kind of scoring scheme. (Statements to the effect that "Product *P* is *x* percent of a good business rule system" are not only absurd but positively harmful, in this writer's very firm opinion.)
- There is no claim that the prescriptions that follow are complete or exhaustive in any absolute sense. Indeed, they are meant to be open-ended, in general. That is to say, anything not explicitly prescribed is permitted, unless it clashes with something explicitly prohibited; likewise, anything not explicitly prohibited is permitted too, unless it clashes with something explicitly prescribed.

The list of prescriptions follows immediately.

> *Note: Some of those prescriptions (the first two in particular) just repeat certain assumptions already spelled out above; however, the assumptions in question are sufficiently important as to deserve elevation to the status of prescriptions per se. The final section of the paper (before the acknowledgments) presents a summary of the entire set of prescriptions.*

1. Executable Rules

Rules shall be **compilable**—see Prescription 10—and **executable** (even when the rule in question is basically just a data definition rule, as in the case of, e.g., CREATE TABLE in SQL).

2. Declarative Rules

Rules shall be stated **declaratively**. The *rule language*—i.e., the language or interface that supports such declarative specifications—shall be **expressively complete**; that is, it shall be at least as powerful as a sorted, two-valued, first-order predicate logic. To elaborate:

- "Sorted" here might better be t*yped* (but *sorted* is the term logicians use for the concept). It refers to the fact that any given placeholder in any given predicate must be of some specific "sort" or type; i.e., its permitted values must be exactly the values that make up some specific data type (or domain—the terms *type* and *domain* mean exactly the same thing [1,2], and they are used interchangeably in this paper).
- "Two-valued" refers to the two truth values *true* and *false*.
- "First-order" refers to the fact that (as already stated under the first bullet above) any given placeholder in any given predicate must take its values from, specifically, the set of values that make up some data type, not, e.g., the set of values—i.e., the set of *relation* values—that is the set of relations currently appearing in the database.

In addition, the rule language shall be constructed according to well-established principles of **good language design**, as documented in, e.g., reference [2]. In particular, it shall exhibit both (a) **syntactic** and **semantic consistency** and (b) **conceptual integrity** (again, see reference [2] for elaboration of these desirable properties).

[1] Especially since, as it happens, the number of prescriptions is exactly *twelve.*

3. Kinds of Rules

Rules shall be loosely divisible into three kinds, as follows:

- **Presentation rules**, which have to do with interactions with the application user (they include rules for displaying interactive forms to the user, rules for accepting filled-out forms from the user, rules for controlling form transitions, rules for displaying error messages to the user, and so forth);
- **Database rules**, which have to do with defining database data, retrieving and updating database data in response to user requests and user entries on interactive forms, specifying legal values for such database data, and so forth;
- **Application rules**, which have to do with the processing the application needs to carry out in order to implement the enterprise work item. (Application rules are sometimes referred to as *business*—or *application—logic*, but these terms are deprecated and not used further in this paper.)

> *Note: It is not always easy to say whether a given rule is a database rule or an application rule, which is why the foregoing categorization is proposed as a loose one only.*

Database and application rules shall include **computations, constraints,** and **inference rules**.

- A **computation** is an expression to be evaluated. The result can be named or can be assigned to one or more variables.
- A **constraint**—frequently referred to more specifically as an **integrity** constraint—is a truth-valued expression (also known as a *conditional, logical,* or *boolean* expression) that, given values for any variables mentioned in the expression, is required to evaluate to *true*.
- An **inference rule** is a statement of the form $p \vdash q$ (where p and q are truth-valued expressions and "$\vdash$"—sometimes pronounced "turnstile"—is a metalinguistic or metalogical operator) that, given the truth of p, allows the truth of q to be inferred.

> *Note: If q is regarded, as it clearly can be, as a truth-valued function, then the inference rule can be regarded as a (not necessarily complete) definition of that function.*

By the way, inference rules as just defined should not be confused with truth-valued expressions of the form $p \Rightarrow q$ (where p and q are truth-valued expressions and "$\Rightarrow$"—sometimes pronounced "implies"—is the *material implication* operator of predicate logic). The expression $p \Rightarrow q$ is defined to evaluate to *true* if and only if p is *false* or q is *true* (equivalently, it evaluates to *false* if and only if p is true and q is *false*). Such an expression might constitute an integrity constraint (e.g., "*e* is an accountant $\Rightarrow$ *e* earns more than $45,000 a year"). By contrast, an inference rule actually *defines q* in terms of p (e.g., "*e* is an accountant $\vdash$ *e* is a white-collar worker").

The following edited extract from reference [4] might help to clarify the foregoing distinction: "Do not confuse $\vdash$ with $\Rightarrow$. The sign $\vdash$ is a symbol of the metalanguage, the language in which we talk about the language that the formation rules have to do with. It makes sense to speak of the formula $p \Rightarrow q$ as being *true* in some state of affairs or *false* in some other state of affairs. However, it makes no sense to speak of $p \vdash q$ as being *true in such-and-such state of affairs;* the truth of $p \vdash q$ has nothing to with states of affairs but with whether the system of logic in which we are operating allows us to infer q from p."[2] (A *state of affairs* can be thought of, loosely, as an assignment of truth values to available atomic formulas.)

The foregoing discussion notwithstanding, it might be desirable in practice (for reasons of user-friendliness, perhaps) for inference rules and material implication expressions both to use the same syntax, *viz*., IF p THEN q. Thus, "IF *e* is an accountant THEN *e* is a white-collar worker" could be an inference rule, asserting that the fact that *e* is a white-collar worker can validly be inferred from the fact that *e* is an accountant. By contrast, "IF *e* is an accountant THEN *e* earns more than $45,000 a year" could be an integrity constraint, asserting that the database must show *e*'s salary as being more than $45,000 a year if *e* is an accountant (and any update operation that would cause this constraint to be violated must be rejected). Of course, if inference and material implication do both use the same syntax, then context will have to provide a means of distinguishing between the two.

Database and application rules might possibly also include **stimulus/response rules**—i.e., rules of the form IF p THEN DO q, where p is a truth-valued expression and q is an action (of arbitrary complexity, in general; e.g., "send a message to the customer" might be a valid action, in suitable circumstances). However, such rules at least partially violate Prescription 2, inasmuch as they are at least partially procedural, and they should be used sparingly and with caution.

> *Note: Stimulus/response rules correspond to what are more frequently referred to as triggers. The idea is that the triggered action q is to be carried out whenever the triggering event p (meaning "p is true") occurs. The keyword IF might more appropriately be spelled WHEN in some situations.*

[2] Yet another way of thinking about the statement "$p \vdash q$" is as a proof: "q is provable from p."

By the way, it is not necessary that rules be "atomic" in any sense, at least from the user's point of view. That is, if *p* and *q* are business rules, then (e.g.) *p* AND *q* is a business rule too. (Rule atomicity might be important from the point of view of the underlying theory or from the point of view of some implementation or both, but it should not be of much concern to the user.)

Rules shall impose **no artificial boundary** between the database and main memory (i.e., the user shall not be required to be aware that the database and main memory constitute different levels of the storage hierarchy under the covers).

4. Declaration Sequence *vs.* Execution Sequence

Business rules will depend on one another, in general, in a variety of different ways; for example, rule *A* might refer to a data item that is defined via rule *B*. This fact notwithstanding, it shall be possible to declare the rules in **any physical sequence**. (Equivalently, it shall be possible to change the sequence in which the rules are physically declared without affecting the meaning.) Determining the sequence in which the rules are to be executed ("fired") shall be the responsibility of the rule engine solely.

Observe that this prescription implies that inserting a new rule or updating or deleting an existing rule will require the rule engine to recompute the rule execution sequence, in general.

5. The Rule Engine Is a DBMS

Database and application rules, at least (and to some extent presentation rules as well), are all expressed in terms of constructs in **the database schema**. Logically speaking, in fact, they are an integral part of that schema; indeed, it could be argued that the schema *is* the rules, nothing more and nothing less. It follows that the rule engine is just a special kind of database management system (DBMS), and rules *per se* are just a special kind of data (or metadata, rather). By virtue of Prescription 10, however, that DBMS can be thought of as operating at some kind of "middleware" level within the overall system; in other words, it is a DBMS that is at least capable of using other DBMSs and/or file systems to hold its stored data (thereby effectively running "on top of" those other DBMSs and/or file systems—possibly several such at the same time).

> *Note: As already indicated, this last point is further elaborated under Prescription 10.*

As just stated, "rules are data" (*database* data, to be precise). It follows that the well-known external *vs.* conceptual *vs.* internal distinctions apply (thanks to Ron Ross for drawing my attention to this point). To be more specific:

- The external form of a given rule is the source form of that rule (i.e., the form in which it is originally stated to the rule engine by the rule definer).
- The conceptual form is a canonical representation of the rule, perhaps as one or more statements of pure predicate logic (a formalism that might not be suitable at the external level for ergonomic reasons).
- And the internal form is whatever form—or forms, plural—the rule engine finds it convenient to keep the rule in for storage and execution purposes.

These three levels shall be rigidly distinguished and not confused with one another.

The external and conceptual versions of any given rule shall include absolutely nothing that relates to, or is logically affected by, the internal (or *physical* or *storage*) level of the system. In particular, those versions shall include nothing that has to do with **performance**.

Since (to say it again) "rules are data," all of the services that are provided for database data in general—including, e.g., conceptually centralized management, access optimization, physical and logical data independence, and recovery and concurrency controls—shall be provided for rules in particular. In other words, **standard DBMS benefits** shall apply. Here are some specific implications of this point:

- Any given user shall need to be aware only of those rules that are **pertinent to that user** (just as any given user needs to be aware only of that portion of the data in a given database that is pertinent to that user).
- Rules shall be **queryable** and **updatable** (see Prescription 7).
- Rule **consistency** shall be maintained (again, see Prescription 7).
- Rules shall be **sharable** and **reusable** across applications (and *vice versa*—see Prescription 9).

6. The Rule Engine Is a *Relational* DBMS

It is well known—see, e.g., reference [2]—that domains (or types) and relations are together both *necessary* and *sufficient* to represent absolutely any data whatsoever, at least at the conceptual level. It is also well known that the relational operators (join, etc.) are closely related to, and have their foundation in, the discipline of first-order predicate logic; they therefore provide an appropriate formalism for the declarative statement of rules, at least at the conceptual level. It follows that it is necessary and sufficient that the rule engine shall be, specifically, a **relational** DBMS,[3] at least at the conceptual level.

Here are some specific consequences of the foregoing:

- **The Information Principle:** The totality of data in the database shall be represented at the conceptual level by means of relations (and their underlying domains) *only*. To say the same thing in another way, the database at any given time shall consist of a collection of **tuples**; each such tuple shall represent a **proposition** that (a) is an instantiation of the **predicate** corresponding to the relation in which it appears, and (b) is understood by convention to be *true*.
- **The Principle of Interchangeability of Views and Base Relations:** The system shall support relational **views**, which, as far as the user is concerned, "look and feel" exactly like base relations; that is, as far as the user is concerned, views shall have *all* and *only* the properties that base relations have (e.g., the property of having at least one candidate key). In particular, views, like base relations, shall be **updatable** [1,2].
- ***Database* is a purely logical concept:** The term *database* refers to the database as perceived by the user, not to any kind of physical construct at the physical storage level. In the extreme, one logical database, as the term is used in this paper, might map to any number of physically stored databases, managed by any number of different DBMSs, running on any number of different machines, supported by any number of different operating systems, and connected together by any number of different communication networks.
- Overall, the rule engine shall function from the user's point of view as **an abstract machine**. It shall not be necessary to go to a lower level of detail in order to explain any part of the functioning of that abstract machine (i.e., the definition of that abstract machine shall be logically self-contained).

7. Rule Management

Since rules are not just data but, more specifically, *metadata*, the portion of the database in which they are held shall conceptually be some kind of *catalog*. It is convenient to refer to that portion of the database as **the rule catalog** specifically. By definition, the rule catalog shall be relational. Suitably authorized users shall be able to access that catalog (for both retrieval and update purposes) by means of their regular data access interface.

> *Note: This prescription does not prohibit the provision of an additional, specialized interface for rule catalog access.*

To the maximum extent logically possible, the rule engine shall:

- Detect and reject **cycles** and **conflicts** in the rule catalog;
- Optimize away **redundancies** in the rule catalog;
- Permit rule catalog updates to be performed **without disruption** to other system activities, concurrent or otherwise.

See also the discussion of "standard DBMS benefits" under Prescription 5.

8. Kinds of Constraints

Integrity constraints shall be divisible into four kinds, as follows [1,2]:

- A **type constraint** is, logically speaking, nothing more nor less than a definition of the set of values that constitute the type (domain) in question. Such a constraint shall not mention any variables.

  > *Note: The constraint that one type is a subtype of another (which provides the basis for supporting inheritance of certain properties from one type to another, of course) is a special kind of type constraint [2]. That is, type constraints shall include what are sometimes called "IS A" constraints (not to be confused with "HAS A" constraints!—again, see reference [2]) as a special case.*

- An **attribute constraint** is a constraint on the values a given attribute is permitted to assume. More precisely, an attribute constraint shall specify that the attribute in question is of a given type (domain).
- A **relation constraint**[4] is a constraint on the values a given relation is permitted to assume. Such a constraint shall be of arbitrary complexity, except that it shall mention exactly one variable (*viz.,* the relation in question).

[3] Not necessarily, and ultimately not even desirably, an SQL DBMS specifically; considered as an attempt at a concrete realization of the constructs of the abstract relational model, SQL is very seriously flawed (again, see, e.g., reference [2]).

[4] More correctly referred to as a *relation variable*, or "*relvar*," constraint (see references [1] and [2]). The term *relation* is unfortunately used in common database parlance to mean sometimes a relation *value*, sometimes a relation *variable* (i.e., a relvar). While this practice can lead to confusion, it is followed in the present paper—somewhat against the writer's better judgment!—for reasons of familiarity.

- A **database constraint** is a constraint on the values a given database is permitted to assume. Such a constraint shall be of arbitrary complexity, except that it shall mention at least two variables, and all variables mentioned shall be relations in the database.

In the case of database and relation constraints (only), the constraint shall also be allowed to constrain **transitions** from one value to another. A constraint that is not a transition constraint is a **state** constraint. In no case shall a constraint explicitly mention the update events that need to be monitored in order to enforce the constraint; rather, determining those events shall be the responsibility of the rule engine.

All constraints shall be satisfied at **statement boundaries**. That is, no statement shall leave the database in such a state as to violate any state or transition constraint, loosely speaking.

> *Note: See reference [2] for a more precise statement of this requirement—in particular, for a precise explanation of exactly what it is that constitutes a "statement" here.*

As well as enforcing constraints, the rule engine shall make its best attempt to use them for purposes of **semantic optimization** [1].

9. Extensibility

Within any given rule, it shall be possible to invoke existing applications[5] as if they were builtin operators. In other words, the system shall be **extensible**, and applications, like rules, shall be **sharable** and **reusable**.

10. Platform Independence

The rule language, and rules expressed in that language, shall be **independent** of specific software or hardware platforms (other than the rule engine itself, possibly).[6] The rule engine shall be responsible (a) for converting —i.e., compiling—rules into executable code that is appropriate to whatever hardware and software environment happens to pertain, and (b) for assigning individual portions of that executable code to processors within that environment appropriately. The concept of platform independence thus embraces at least all of the following (see reference [1]):

- Independence of the overall implementation environment ("system architecture independence");
- Hardware independence;
- Operating system independence;
- Transaction monitor independence;
- Network independence;
- Location, fragmentation, and replication independence;
- DBMS independence.

Regarding the last of these in particular, note that it shall be possible for an application to **span** several distinct backend subsystems, running several distinct DBMSs and/or file systems. It shall also be possible for an application (declaratively built, of course) to access **preexisting** databases and/or files. In all cases, the necessary **mappings** between backend database schemas and the rule engine's own database schema (and/or the rule catalog) shall themselves be specified by means of appropriate business rules.

11. No Subversion

If an interface is supported that provides access to the database at a level below the conceptual level (in effect, below the level of the abstract machine that is the rule engine), then it shall be possible to **prevent use of that interface** for purposes of subverting the system. In particular, it shall be possible to prevent the bypassing of any integrity constraint.

12. Full Automation

Applications developed using business rules shall be **complete** (loosely, "everything automatable shall be automated"). That is, the complete set of rules for all of the applications that are pertinent to the enterprise in question shall constitute a **complete business model**, or in other words an abstract definition of the entire enterprise and its workings. In fact, because rules are shared across applications and *vice versa*—see Prescriptions 5 and 9—the activity of defining rules in general can be seen not so much as a process of developing individual applications, but rather as one of developing **entire integrated application systems**.

---

[5] Refer back to the "Assumptions" section for an explanation of the term application as used in this paper.

[6] It could be argued that this prescription is a straightforward logical consequence of the requirement, already articulated under Prescription 6, that the rule engine function as an abstract machine.

Summary

By way of conclusion, here is a brief (and somewhat simplified) summary of the twelve prescriptions.

1. **Executable Rules:** Rules are compilable and executable.

2. **Declarative Rules:** Rules are stated declaratively. The rule language is well designed and expressively complete.

3. **Kinds of Rules:** Rules are divided into presentation, database, and application rules. They include computations, constraints, inference rules, and possibly stimulus/response rules.

4. **Declaration Sequence *vs.* Execution Sequence:** Rules can be declared in any sequence. The rule engine determines the corresponding execution sequence.

5. **The Rule Engine Is a DBMS:** Rules are expressed in terms of constructs in the database schema. The rule engine is a special kind of DBMS; it can act as middleware, using other DBMSs and/or file systems to hold its stored data (possibly several such subsystems at the same time). Rules exist in external, conceptual, and internal forms; the first two of these, at least, include nothing to do with performance. Rules are shared and reused across applications.

6. **The Rule Engine Is a *Relational* DBMS:** At the conceptual level, at least, the rule engine is relational. It acts from the user's point of view as an abstract machine.

7. **Rule Management:** Rules can be queried and updated. Insofar as is logically possible, the rule engine detects and rejects cycles and conflicts (and optimizes away redundancies) among the rules, and permits rule updates to be done without disrupting other system activities.

8. **Kinds of Constraints:** Constraints are divided into type, attribute, relation, and database constraints. Transition constraints are supported. Constraints are satisfied at statement boundaries.

9. **Extensibility:** Rules can invoke existing applications.

10. **Platform Independence:** The rule engine provides independence of hardware and software platforms. It also provides independence of the overall system architecture, by assigning compiled code to available processors appropriately. Applications can span backend subsystems (possibly preexisting ones).

11. **No Subversion:** The database cannot be accessed below the conceptual level in such a way as to subvert the system.

12. **Full Automation:** The rule engine supports the development of entire integrated application systems.

Acknowledgments

Thanks to Manish Chandra, Hugh Darwen, Mike DeVries, Val Huber, Gary Morgenthaler, Ron Ross, and Gene Wong for helpful comments on earlier drafts of this paper and other technical assistance.

References and Bibliography

1. C. J. Date: *An Introduction to Database Systems* (7th edition). Reading, Mass.: Addison-Wesley (2000).

2. C. J. Date and Hugh Darwen: *Foundation for Future Database Systems: The Third Manifesto* (2nd edition). Reading, Mass.: Addison-Wesley (2000).

3. GUIDE Business Rules Project: *Final Report*, revision 1.2 (October 1997).

4. James D. McCawley: *Everything that Linguists Have Always Wanted to Know about Logic (but were ashamed to ask).* Chicago, Ill.: University of Chicago Press (1981).