Model-Based Interchange Formats: a Generic Set of Tools for Validating Structured Data against a Knowledge Base

Pascal Rivière[1], Olivier Rosec[2]

[1]Head of Methodology Dept.
[2]Head of Social Data Interchange Unit
(within the Methodology Dept.)

Caisse nationale d'assurance vieillesse (Cnav)
110 – 112 avenue de Flandre F-75019 Paris

Abstract -A Opting for a model-based approach to develop a set of tools for validating structured data concentrated at the beginning on the generic control engine which would read a knowledge base containing rules. But to attain this goal, one had to develop a Model Editor which over time evolved into a full-fledged Integrated Development Environment (IDE) for the modeling of structured data formats, the specification of their validating rules and the generation of the knowledge base.

A) Introduction

In this era of fine-grain interactions between complex systems across distributed architectures, file processing has acquired a somewhat quaint flavor. But even nowadays there is no other way to transmit complex data from one system to another.

Difficulties are many at all stages:

- Defining the interchange format between partners is not easy;
- Ensuring the actual files meet the quality standards expected in production means multiplying control rules;
- Defining the architecture, protocols and syntax for file processing platforms adds yet another dimension to the conundrum.

This paper is about the search for a generic approach covering the first two points.

B) Problem Domain: Rationalizing the Social Data Collection Format

In France social protection is split up in several schemes administered by different agencies. Over the course of time each agency has discovered the hard way that it is less costly to acquire data about the future claimants of benefits in a steady stream, at the source, directly from the payroll system, rather than on an *ad-hoc* case per case basis, from the claimant.

Thus all agencies which pay out old age pensions on the basis of contributions paid along one's working life have turned to collecting the data about employees' pay on a yearly basis, instead of collecting it from the faded pay slips of a lifetime when the employee claims his or her pension.

Social data, as a broad term covering pay-related data, also serves to test whether an employee is entitled to this or that benefit, be it a sickness benefit or an unemployment benefit.

Other services outside the social protection sphere have been interested as well: the Inland Revenue, the National Office of Statistics want to use that kind of "big data" either for sending out tax forms pre-printed with income returns or to conduct surveys.

Initially the data collection process relied on paper forms. The paper forms merged into a single one, and as the process went digital during the 1980s, that huge form gave birth to a file format. The interchange medium switched from tapes and diskettes to the Web around year 2000. Today, more than one million employers send files at the beginning of each year.

The success of that particular community has attracted more and more partners, because once a reliable channel for the transmission of data from payroll systems to the information systems of public services has been found, it is far easier to plug into it than to set up a brand new one from scratch.

A new standards body was set up in 2008 to organize the process of collecting requirements beyond the original community of partners. But the standards body has no leverage whatsoever on the data collection process: the data is in fact distributed over a series of platforms. It cascades through a complex splitting and filtering process so that each administration gets the data relevant to its business purposes and just that. And some partners insist on running their own platform.

C) The Interchange Format as a Maintenance Nightmare

The **interchange format** is represented **as a hierarchy of data blocks governed by an alphanumeric naming scheme**.

At file level, data elements are physically represented on a key-value basis, the key being the identifier of the data element according to the naming scheme. There is no physical notation of data element blocks. The naming scheme enforces the model organization within the flat file format.

Separators and an end of line character are other precisions given in the file format specification, as well as the character encoding, with restrictions for particular data elements.

There is no typing other than alphanumeric, numeric, date. Typing can be further refined by regular expressions and minimum and maximum lengths. Some data elements have to belong to a list of values defined as an enumeration or carried by an external referential.

Control rules, written out in natural language, describe consistency checks between data elements: co-occurrence, comparison tests enforce semantic validity at file level.

Yearly Change Requests : the Maintenance Challenge

Each year nearly one thousand change requests are introduced by partners because of:

- changes in legislation;
- "patches" to solve production issues arisen during the last data collection campaign.

The national agency in charge of the format must then update the file specification and each team must update the corresponding application code on their data-processing platform.

The frequency of change requests has created a maintenance challenge which is further aggravated by the following facts:

- The specification is considered as a document to be discussed during countless proof-reading sessions;
- The focus, instead on being on concepts, is on implementation details. There is no proper conceptual data model independent of the file format. There are only broad rules governing the organization of data blocks carrying data elements along several axes:
    - A semantic axis along which one finds in succession the description of the party sending the file, of the employer, the employee and the business data for this employee;

- A temporal axis which governs the insertion of working periods for an employee within the timeframe carried by the file: month, quarter, year;
- An "ownership" axis because business data is split between "common" business data received by all partners and business data specified by and "belonging" to a particular partner.

Administering the format specification along those three axes gives birth to one of those combinatory explosions which go hand in hand with a requirements elicitation process chugging along contentedly in chronic happy-hour mode. The maintenance challenge turns into a nightmare.

The Stand for a Generic Model-Based Approach

The national agency in charge of the file format and historically responsible for the main file processing platform got fed up with:

- The absurdity of writing specific hand-crafted code which had to be thrown away each year as the file format specification evolved;
- Squabbles between developing teams over the interpretation of this or that rule;
- The slow turnaround time when a control program had to be patched.

It made a stand in favor of a generic approach and took a step further the breakaway from a mere paper specification**. From a single referential** which would represent the file format, **one should be able to generate:**

- **The documentation** for implementing it across the community;
- **A knowledge base.**

The knowledge base would be read by a generic engine which would execute all rules. The engine would remain the same over the years. Only the knowledge base would change.

The whole specification would become machine executable. A team would take care of the modeling which would produce both documentation and knowledge base. No more code, no more developers. But first one had to jump over a few hurdles.

D) "Abstract Implementation": Domain-Specific Languages

To enable the design and development of a suite of tools addressing the needs of the modeling team in charge of the file documentation and knowledge base, first one had to lay the foundations:

- Meta-models for the file format and deliverables;

- And transformation strategies to be applied to the single referential persisting the models, to generate the deliverables.

The software solution has been designed on the basis of a Domain-Specific Language.

"A DSL is a programming language tailored specifically to an application domain: rather than being for a general purpose, it captures precisely the domain's semantics. (...) DSLs allow the concise description of an application's logic reducing the semantic distance between the problem and the program." [Spinellis, 2000].

Each time we can, we will use Spinellis's taxonomy of patterns in the remainder of this paper to explain the way a DSL supports the software process which is being described.

The priority for the problem domain was to design the data model from which interchange formats would be built. The model articulates three libraries:

- A **Structures library** describing data blocks composed of data elements;
- A **Data types library** describing the types for data elements;
- A **Messages library** describing each interchange format as a hierarchy of data blocks.

The three libraries persist the current data interchange format modeled with the help of the meta-model. This corresponds to the data structure representation creational pattern [Spinellis, 2000].

Data block properties include:

- An identifier composed according to the naming scheme;
- A functional name;
- A description;
- A multiplicity (there can be 0, 1 or N instances of each block).

Data element properties include:

- An identifier composed according to the naming scheme;
- A functional name;
- A description;
- A usage (each data element can be within a certain block mandatory, conditional, optional, or forbidden).

Rules are attached to data elements. Block level rules are attached to the first data element in the block. Rules properties include:

- An identifier;
- An execution context;
- A message to be returned to the user in case the rule is triggered and not satisfied;
- The rule in natural language.

Semantic rules have been represented by a **textual DSL** which was first specified in EBNF. The rules are written as **mathematical propositions enforcing first-order predicate logic**. They can include **existential or universal quantifiers**. Semantic rules are written using the fully qualified identifiers for the data elements. Thus they are easily read and debugged.

Semantic rules can call **macros** and **aliases**. Both can be used as shorthand to simplify a complex rule: for example, does this employee belong to the public sector and if it is true then execute B and if not execute C. Semantic rules can be extended by **functions** mapped to the function prototype of an executable language.

Documentation has been modeled too. A file format specification is a document consisting of:

- Resources which are references to static document or spreadsheet formats;
- Templates for exploring the referential, through a reporting engine which will bring back the selected objects: messages, data blocks with their elements and types and rules.

The DSL which federates the resources and parameters for documentation generation illustrates the system front-end DSL pattern [Spinellis, 2000].

E) “Concrete Implementation”: The Eclipse Modeling Framework

EMF’s main “selling point” (it is for the most part open source and free) is that it is built on top of the Eclipse platform. The Eclipse platform is in itself an asset, providing countless mechanisms and wizards for managing projects, writing, compiling and debugging code, managing code libraries and source repositories, tracing file change. It plugs into most source control and ticketing tools.

EMF started according to the literature [Merks, Gronback, 2009] as a reaction against the profuseness of the Unified Modeling Language. A subset of UML constructs called Ecore articulates the minimum set of components to build models from EElements, EClasses, EAttributes etc.

EMF enables one to build such a model, from scratch through the appropriate editor, or through the transformation of:

- A UML model;
- Annotated Java code;
- XML Schema.

The Model Editor

With EMF one can build quickly an editor to manipulate business models. A powerful API helps enforce Model View Controller (MVC) and command stack mechanisms. Models can be persisted as resources in an XMI style syntax. Various template engines are available for model to model or model to text transformations.

These transformations combine the source-to-source transformation creational pattern and the pipeline behavioral pattern [Spinellis, 2000].

Over three years the File Format Editor has gone through many different versions as models were refined and deliverables tuned to the needs of the user community.

Model resources have been organized into a model bundle within which a catalog file points to all resources such as the three aforementioned libraries.

The same models go into the making of the knowledge base which is compiled as a Java project and organized in directories read by the control engine as it goes through its different processing stages.

Automatic generation reduces turnaround time to deliver a new knowledge base to one hour, including non-regression tests which have been automated (test files reports are parsed to compare the obtained result with the expected result), once for instance a rule has been patched.

The Three Representations of a File Format

The file format is modeled in the Editor through a graphical user interface.

The seminal decision was to represent the file formats in XML Schema in the knowledge base. All other decisions hinge on that choice.

XML Schema is a cheap and common way of structuring data. It offers strong typing. An XML instance can be parsed and validated against the schema it purports to respect.

But the actual files remain true to the legacy flat key-value format.

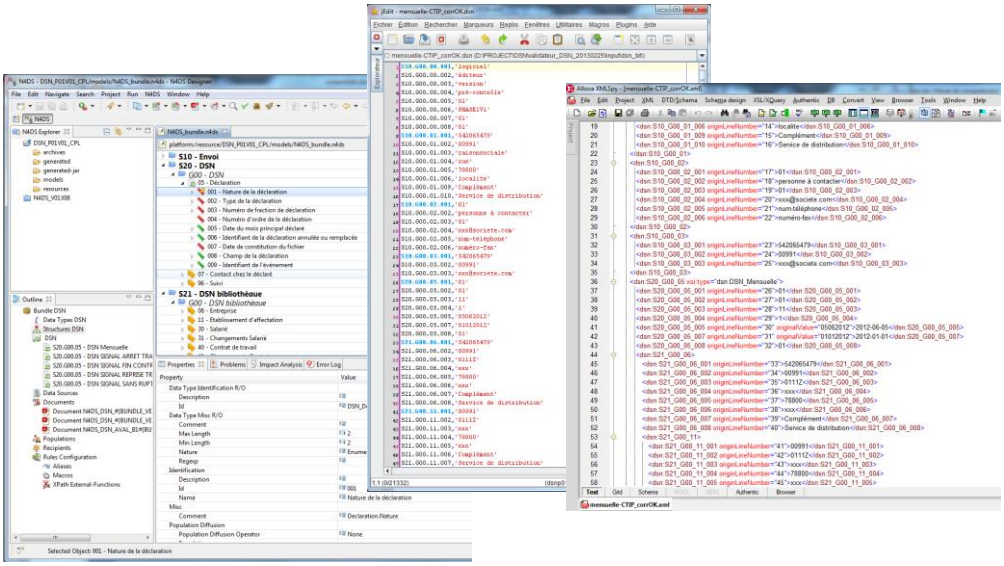

Figure 1 shows side by side the model as seen through the editor, an instance of the flat file legacy format, and its XML conversion.

The Validating Engine and its Processing Stages

The generic control engine processes a file in three stages:

- **Conversion** from the flat key-value legacy format to a hierarchical XML instance;
- **Syntactical control**, by validating the instance XML file against the XML Schemas;
- **Semantic control**, by firing one after the other the rules attached to data elements.

The control logic is static. There is no interface to live databases to check the existence or the status of the value of a data element. Only the knowledge base will be read.

The Three Representations of a Semantic Rule

XML Schema offers no easy way to enforce consistency constraints between data nodes. One has to write specific code. But specificity was not the order of the day. Hence the decision to implement the textual DSL described earlier in the paper.

S21.G00.40.009/CCH-12
The job contract number must be unique for a given employer and employee
DSL

```
every x:S21.G00.40,y:S21.G00.40 satisfies (((($x!=$y) and is_present($x:S21.G00.40.009) and is_present($y:S21.G00.40.009)) =>
(($x:S21.G00.40.009 != $y:S21.G00.40.009))
```

Java

```
 public IRuleResult run(ISousGroupe context)
 {
  S21_G00_30 ctxt = (S21_G00_30)context;

  S21_G00_30 var_root = ctxt;

  Iterable s21_G00_40 = ctxt == null ? Collections.EMPTY_LIST : Iterables.filter(Iterables.concat(new Iterable[] { ctxt.getS21_G00_40() }),
Predicates.notNull());

  Iterable _s21_G00_40 = ctxt == null ? Collections.EMPTY_LIST : Iterables.filter(Iterables.concat(new Iterable[] { ctxt.getS21_G00_40() }),
Predicates.notNull());

  Iterable s21_G00_40_009 = Iterables.filter(Iterables.concat(new Iterable[] {
   Iterables.transform(_s21_G00_40,
   new Function()
  {
   public S21_G00_40_009 apply(S21_G00_40 arg0) {
    return arg0.getS21_G00_40_009();
   }
  }) }), Predicates.notNull());

  IRuleResult ruleResult = null;
  Boolean result = Boolean.valueOf(false);

  result =
   Boolean.valueOf(Operators.every(s21_G00_40, new Predicate(s21_G00_40)
  {
   public boolean apply(S21_G00_40 var_x) {
    boolean result =
     Operators.every(this.val$s21_G00_40, new Predicate(var_x)
    {
     public boolean apply(S21_G00_40 var_y) {
      boolean result =
       (this.val$var_x != var_y) &&
       (ExternalFunctions.is_present(RuleS21_G00_40_009_CCH_12.this.s21_G00_40_009From(this.val$var_x))) &&
       (ExternalFunctions.is_present(RuleS21_G00_40_009_CCH_12.this.s21_G00_40_009From(var_y))) ?
       Operators.neq(RuleS21_G00_40_009_CCH_12.this.s21_G00_40_009From(this.val$var_x),
RuleS21_G00_40_009_CCH_12.this.s21_G00_40_009From(var_y)) : true;
      return result;
     }
    });
    return result;
   }
  }));
```

Figure 2 shows the transcription of a rule from the paper specification to its code implementation in the knowledge base (Java code excerpt here), obtained from a parsing of the textual DSL.

Another problem dogging the processing of big XML files is memory management which opens up on the usual alternative: event-driven parsing (SAX) or document loading (DOM). In the time-honored way of hand-crafting control code, one positions control rules involving variables belonging to data blocks stretching across the whole file when the last necessary variable will have been read and stored.

The original vision wanted to dispense with the turnaround time associated with hand-crafted code. So the parser which transforms textual DSL had to transform it into machine executable code supporting:

- The test logic which would return a Boolean;
- The data addressing mechanism;
- And ultimately a memory-management mechanism.

A semantic validation API in Java covers all three issues, and more specifically memory-management through a twin set of utility classes loading and unloading variables as the engine fires rule after rule to check a file, however big it may be. The API rests on the convention that data elements always have the same address, the one they have in the "covering message" which is a superset of all messages within the model.

Hosting a semantic rule API in Java corresponds to the piggyback structural pattern [Spinellis, 2000].

F) The Validating Engine in Real Life

One should speak less in terms of an implementation gap and more in terms of a consistent way of dealing with the issues which arose in the course of the project and which had to be solved on the spur of the moment as the product neared roll-out time in late 2012.

The Project Cycle

If one adopts the Y shape used to describe the fusion of the upper branches carrying business requirements and system-level frameworks into an end-product, one should say that the work cycle, instead of trickling down the Y, more or less pulsated in radiating circles from the middle of the Y, as, from version to version, the set of implemented functionalities and the range of transformation strategies and frameworks used to develop the product expanded from the original nucleus.

But there are issues associated with deployment which can be addressed only with the help of real user and qualification team feedback. This feedback accounts for the A shape superimposed on the Y shape. The A shape denotes:

- Deployment issues such as performance, ease of integration;

- Usability in terms of user-friendliness, which means reducing the distance between the original file and the converted file processed by the control engine, by keeping as attributes:
    - the original value of certain data elements transformed from the legacy string format to comply with one of XML Schema's built-in datatypes (for example, dates);
    - the line number of the data element in the original file.

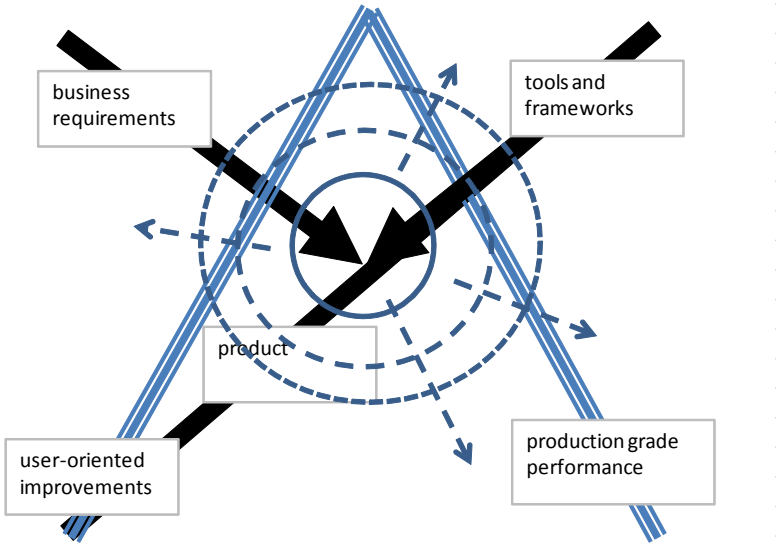

Figure 3 superimposes the Y (development) and A (production and user feedback) cycles

Stateless Mode and Report Stream Related Issues

Processing files in a production environment means processing gracefully even badly damaged files, to return a user oriented report and not just a log trace. And the user community wants validation reports to be exhaustive to understand what was wrong with the file and the system it comes from. The control engine is **stateless** and goes from one stage to another even if errors were detected at an earlier stage. But errors at an early stage provoke errors at later stages: the report gets more and more confusing for the user.

It might prove more efficient in the future to stop processing files at a certain stage. This could mean redesigning the report stream which is open and closed at each stage (intermediate reports are then merged into a full report). A continuous report stream could be a better solution and would provide the interface necessary to stop file processing before the user report loses all relevancy.

G) Return on investment

Originally the suite of tools was developed to support the **Norme pour la Dématérialisation des Déclarations de Données Sociales** (N4DS: 800 data elements, 600 semantic rules). It now supports the **Déclaration Sociale Nominative** (DSN: 400 data elements, 120 semantic rules) as well, with no fork in the code of both Editor and Engine. Since the roll-out of the first DSN validating component, numerous releases have been made, including several emergency knowledge base patches within half a day. This would have been impossible with hand-crafted code.

References

Diomidis Spinellis, « Notable Design Patterns for Domain-Specific Languages », Journal of Systems and Software, vol. 56, no 1, 2001, p. 91-99

Dave Steinberg, Frank Budinski, Marcela Paternostro, Ed Merks, "EMF Eclipse Modeling Framework", Addison-Wesley, Pearson Education, 2009.

Richard C Gronback, "Eclipse Modeling Project, a Domain-Specific Language (DSL) Toolkit", Addison-Wesley, Pearson Education, 2009.