Domain-Specific Languages in a Customs Information System

Margus Freudenthal, *Cybernetica AS*

Cybernetica AS's Customs Engine comprises several subsystems built on a componentized platform via domain-specific languages, enabling a highly iterative and reuse-oriented development method.

Through its use of appropriate notations and abstractions, a domain-specific language (DSL) offers expressive power focused on—and usually restricted to—a particular domain.[1] In other words, DSLs trade broad applicability for expressive power in a particular area. Classic examples include Unix makefiles (build scripts), regular expressions (for specifying text patterns), HTML (for describing text layout), and GraphViz (for describing graphs).

The literature offers many descriptions of the benefits of using DSLs, including improved development productivity, flexibility, maintainability, and separation of business and technical aspects. For instance, Don Batory and his colleagues report on their use of DSLs to implement a command-and-control simulator.[2] Their case study demonstrated added development productivity and flexibility with respect to a Java implementation. Arie van Deursen and Paul Klint reported using a DSL in the financial engineering domain with an emphasis on added maintainability.[3] Finally, Satish Chandra and his colleagues and Scott Thibault and his colleagues discuss case studies in which they used DSLs to implement video device drivers and distributed cache coherence protocols, respectively, demonstrating the benefits of separating technical aspects from business logic.[4,5] However, in spite of all this work, little evidence supports the applicability of DSLs in large-scale information systems development. This article addresses that gap by reporting on a successful case study in which DSLs served as a customs information system's backbone.

Customs Engine

Since 2005, Cybernetica AS has worked with the Estonian Tax and Customs Board in building Customs Engine (CuE), a suite of systems for processing customs documents (such as declarations, manifests, warehousing notices, and so on). Each type of document reflects a movement of goods—for example, between a ship and a warehouse—and is associated with its own set of rules, regulations, and procedures. Given the domain's document-heavy focus, the CuE development team (of which I was a part of) decided to decompose CuE into subsystems, such that each one is responsible for processing one type of document.

CuE runs on standard Java EE application servers. Much like Microsoft Office contains programs that are useful even if the rest of the suite isn't installed, each CuE subsystem is an independent application that can run separately. In addition to providing a user interface for manipulating certain documents, CuE subsystems communicate with other CuE subsystems and their counterparts in the EU to track movement of goods across several

 0740-7459/10/$26.00 © 2010 IEEE

customs procedures. CuE follows service-oriented architecture (SOA) principles, meaning its subsystems send notification messages and implement services that other applications can call.

These subsystems have functional and technical similarities, so we adopted a software product line engineering approach to take advantage of them. Our decision to use DSLs in developing CuE was based on the following concerns:

- Because of the significant variability among different CuE subsystems, the platform's components must be highly flexible and configurable.
- CuE implements sizable and complex legislation and working procedures that change over time, so it's important to implement processing rules in a way that provides a clear overview of what the system does. Preferably, updating the rules shouldn't require additional programming, rebuilding, or restarting the system.

When we started designing the first subsystems, we anticipated the complex and volatile nature of the business logic required for verifying the correctness of submitted customs documents (together with checks to external registries and calculation of some fields, such as taxes). This complexity stems from the fact that customs is very heavily regulated in the EU, with extensive standards and regulations concerning each type of document. The volatility comes from two sources. First, the EU requirements were new both to us and the domain specialists in Estonian Customs (we started building CuE shortly after Estonia's accession to the EU). This implied iterative development of the business logic with continuous user feedback to ensure that system functionality corresponded to actual working procedures. Second, both legislation and customs procedures change over time, implying that the system must maintain verification rule histories (older documents are verified by rules in effect at the time the document was originally submitted).

We chose to encapsulate the document verification logic into a separate component that receives the document to be checked (in XML format) and returns a list of verification results (errors, warnings, or tasks for the customs officer). Verification rules aren't hard-coded into the component—rather, they're programmed in a DSL. The verification component contains an implementation of the document verification DSL. Although it could have been possible to implement this component using an off-the-shelf rule engine (such as Drools; www.jboss.org/drools), we decided to implement a custom document verification language based on the following considerations:

- We predicted that the amount of verification rules would be very large (each subsystem has its own set of verification rules), so the ease of rule development might offset the costs of developing a custom verification rule language.
- Upon analyzing the rules, we discovered that most of them follow one of a few recurring patterns. We therefore decided to implement direct support for these patterns so that, for simple rules, the system did the "right thing" automatically.
- We wanted to have tight integration among the user interface for document creation, the verification module, and the other verification steps (such as simply checking the presence of mandatory fields or whether a field's value belonged to the correct code list). This tight integration lets the user interface provide precise visual feedback about fields containing verification errors or warnings.

Building on our experience, we started structuring most of our main components into two layers: the higher layer has a formal specification of what the component does (written in a high-level DSL), and the lower layer is the DSL's implementation. Our aim was to make the specifications in the higher layer concise, human-readable, and nontechnical.

In general, our primary goal wasn't to make the executable specification so detailed that it fully captured the component's behavior. When trying to create comprehensive DSLs, we often encountered some extremely complicated cases that required a powerful and complex language that would have resembled a general programming language and thus sacrificed most of the advantages gained by focusing on a specific domain. Therefore, instead of trying to express all the details via a DSL, we followed an 80-20 approach and focused on brevity and clarity. We used the DSL to give an overview of the component's behavior (describing its "essence") and solved the complicated corner cases by implementing them in Java and calling the Java code from the DSL. This kept our DSLs simple and readable.

Domain-Specific Languages in CuE

CuE relies on a DSL that can express document verification rules in the customs domain (namely, Burula) and several configuration DSLs.

Figure 1. A simple verification rule. A natural language would express it as "If the goods are unpacked (indicated by using codes NE–unpacked, NF–unpacked, 1 item, **or** NG–unpacked, several items**), then the** number of pieces **field must be filled in."**

```
predicate is-unpacked-goods
    kindOfPackages is ('NE', 'NF', 'NG')
packages must have numberOfPieces
        when is-unpacked-goods
    error "When goods are unpacked, number
        of pieces must be present"
```

Burula

We used the Burula programming language to specify rules for verifying the correctness of the documents submitted to the system. The input to the verification process is the document in XML format, and the output is a list of verification errors; each error contains a human-readable description and the erroneous value's location.

Burula is the result of several years of iterative development. The team that developed the CuE designed the initial versions in an ad hoc manner, without much knowledge about what the verification rules would require. We based the development on our own experiences with other verification languages, which helped us optimize for the most common case.

The most typical type of correctness check concerns dependencies between fields, following the pattern "if field *X* contains value '*foo*,' field *Y* must contain value '*bar*'" (where '*foo*' and '*bar*' can be sets of possible values or simply a value's presence or absence). Roughly 80 to 90 percent of all checks fall into this group—the rest of the verification involves comparing the submitted document with other data, such as previous versions of the same document, licenses, permits, various reference data, and so on. This other data usually resides in other systems and must be queried via various interfaces.

When designing Burula, we intended for analysts and domain specialists at the customs board to write or modify the verification rules. From this goal, we derived the following main requirements for the verification rules language:

- The rules must be structurally similar to the rules expressed in a natural language.
- The rules must contain a minimal amount of technical information—for example, referring to document fields must be possible without spelling out the exact location of fields (the user can use packages instead of declaration.goodsItem.packages). Additionally, iteration over repeated elements should occur transparently.
- The language must have strong static type checking.
- The rule language must enforce a good style. In particular, it should discourage writing long, complex rules with complicated Boolean expressions containing unexplained "magic" values.
- Because the verification rules change more frequently than the core business logic, it must be possible to change the rules without restarting the system. Additionally, rules must be versioned and old documents verified via the rules in effect at the time the document was originally submitted.

Figure 1 contains an example of a business rule written in Burula. A natural language would express this rule as "If the goods are unpacked (indicated by using codes NE–unpacked, NF–unpacked, 1 item, or NG–unpacked, several items), then the number of pieces field must be filled in." (For packed goods, the declarant fills the number of packages field instead.) Note that this check applies to all the package descriptions in all the goods items of the declaration.

This rule takes advantage of Burula's implicit iteration. The Burula implementation automatically iterates over all the goods items and all the package descriptions inside those items; it also applies the predicate is-unpacked-goods only to the package description that the rule currently examines. Moreover, this example shows how Burula tries to enforce good writing style by restricting rule complexity: the rule author can't use the condition on the kindOfPackages field directly as part of the rule and must instead write it as a separate predicate, ensuring that it will have a name (and thus contain information about its purpose).

Figure 2 shows a more complicated rule that checks whether the export movement's itinerary contains more than one country. XmlEcs is the name of the root element of the XML document representing an export report. It indicates that this rule's scope is the whole document (as opposed to the example in Figure 1, where the scope is one package description). Additionally, the Burula implementation can use it to determine the verification error's location (all the generated verification errors contain pointers to the field containing the error).

To provide performance comparable to writing rules in Java, Burula compiles its source files to Java bytecode (class files). An additional benefit is improved integration with the rest of the system—for example, it's easy to call Burula programs from Java code and Java methods from Burula. The Burula implementation performs the helper function calls in the same execution context as the rest of the system and the Burula rule. Compiled Burula

programs are saved in a database and loaded when the system runs the verification rules, which lets the rule author modify those rules without restarting the system.

The Burula toolset is currently quite Spartan and consists of the following items:

- a Burula compiler and runner embedded in each CuE subsystem;
- a stand-alone compiler for testing verification rules; and
- a text editor (vim; www.vim.org) configured with syntax highlighting support for the Burula language.

The rule author creates and manages the business rules offline by using a standard text editor and Subversion for version management. Initially, we experimented with embedding the rule-editing functionality into the CuE user interface, but we found that our Web-based tools were inconvenient for this purpose and that standard tools were best for version control.

Configuration DSLs

The document verification part of the system contains a bulk of application-dependent business logic. For these parts, we can justify creating a complex, special-purpose DSL implementation, especially if several subsystems built on the same platform can share the implementation costs. We used several smaller configuration DSLs to customize each CuE subsystem's platform components. Compared to the document verification module, these components had some important differences:

- The configuration DSL programs are simpler and contain far fewer rules than Burula programs.
- Although verification rules are almost entirely derived from external specifications and user requirements, configuration DSLs often express technical concerns, such as how to configure a messaging component. Also, configuration DSL expressions are less likely to change during the project's life.
- Configuration DSL expressions are written by technical people, so there's less emphasis on friendly syntax.

Because CuE had many small configuration DSLs, we had to keep the cost of developing each individual language low. We couldn't afford to implement custom compilers for each one, so we decided to implement the languages via a templating engine to generate Java code or XML configuration files. This approach has several benefits:

- We can reuse existing Java compilers and tools, thus lowering implementation cost.
- We inherit features from Java, so by cleverly using Java types in the generated code, we could use Java type checker to verify a DSL program's correctness.
- We can embed Java statements and expressions inside DSL code, which makes the DSL relatively simple and able to support the most important cases. For complex corner cases, we can use Java directly.
- Files input to the templating engine share the same base syntax, which simplifies learning the next configuration DSL.

We reused a previously developed in-house templating engine called Templater that has good performance, support for Java code generation, and good integration with our build system.

Figure 3 shows how Templater works. Basically, the template matches elements in the input file, which contains a program in a DSL, and generates parts of the output file using the

```
predicate is-ship-supplies
   specificCircumstanceIndicator = 'A'
predicate is-postal-consignment
   specificCircumstanceIndicator = 'B'
XmlEcs lengthOf itinerary > 1
   unless is-ship-supplies
      or is-postal-consignment
   error "If the declaration does not contain
      ship supplies or postal consignment, the
      itinerary must contain at least
      two countries."
```

Figure 2. A more complicated Burula rule. XmlEcs **is the name of the root element of the XML document representing an export report. It indicates that this rule's scope is the whole document (as opposed to the example in Figure 1, where the scope is one package description).**

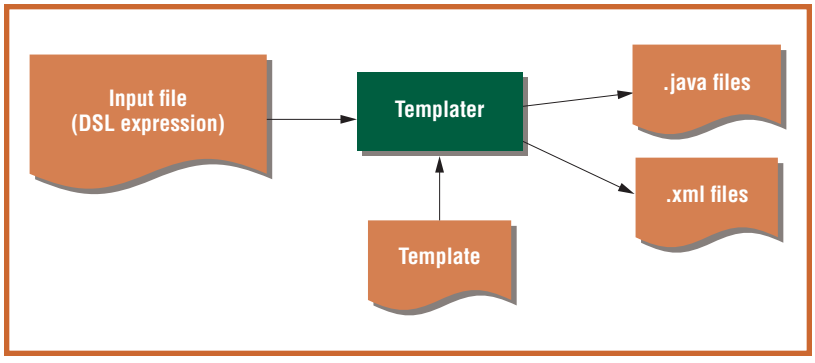

Figure 3. Use of Templater. Templater interprets the template that determines how the input file is transformed to output files.

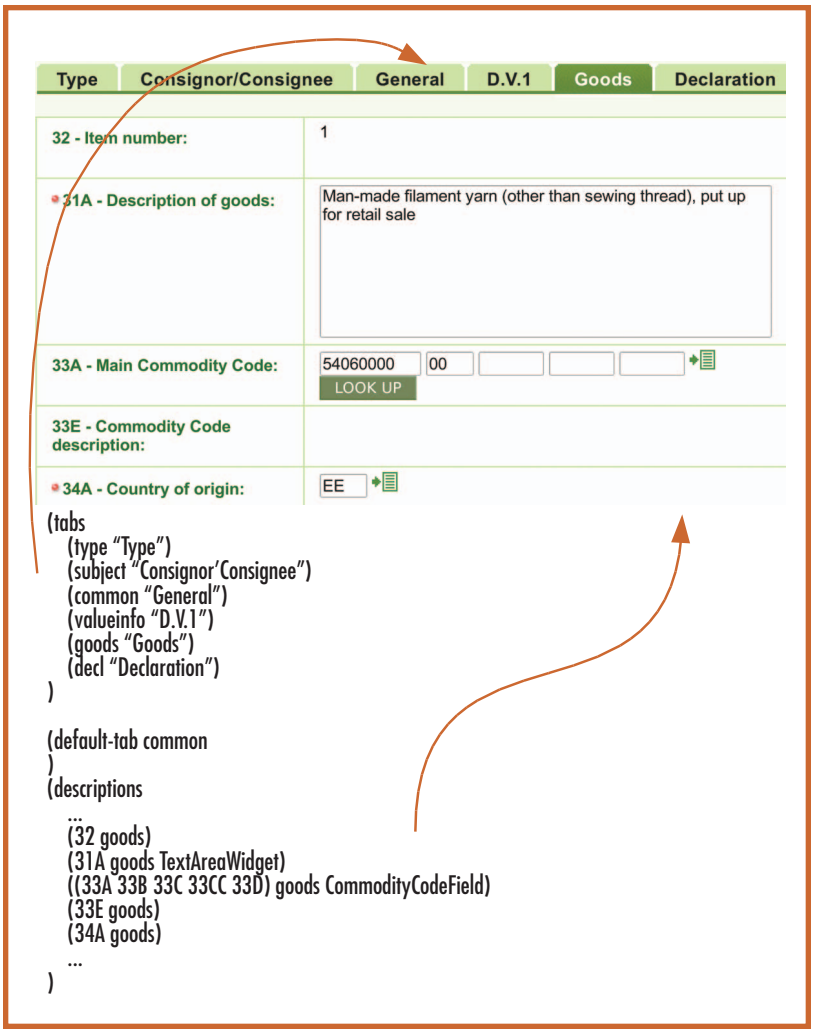

Figure 4. Document-editing screen and its corresponding domain-specific language code. The DSL code describes the layout of the screen.

data contained in the matched input element. The input files have standard syntax based on S-expressions.

Figure 4 shows an excerpt of a configuration DSL expression that describes a document-editing user interface's layout. The document editor uses a tabbed, wizard-like user interface that displays document fields in a form. All the document fields have an alphanumeric code (such as 10, 23A) to make it easy to refer to them in documentation. The excerpt in Figure 4 defines which tabs the editor displays and the document fields that go with those tabs.

Our Experiences

Our overall experience in using DSLs, based on four years of developing and maintaining CuE, is quite positive. CuE subsystems are quite small in code size—typically 5,000 to 25,000 lines of code (including all the code written in DSLs)—and the platform itself is roughly 100,000 lines of code. We attribute the small size to extensive reuse and a raised level of abstraction (we coded parts of the system in high-level DSLs).

From a software architecture standpoint, our extensive use of DSLs forced us to separate the system into "business" and "technical" parts and define explicit interfaces between them. This clearly separated them and let us change one part of the system without noticeably affecting the other. Our experience building CuE also showed that although we could reuse roughly half the platform components simply by instantiating platform classes and calling methods, we had to parameterize the other half with code. The required code was often quite technical and consisted of several related classes and methods, but by using DSLs, we could generate this complicated code from a concise and human-readable DSL program, facilitating platform component reuse.

Besides its impact on the technical architecture, DSLs also influenced the way we work. One notable change is that much of detailed programming work transferred from programmers to analysts, which led to a quicker and more iterative development cycle. Analysts could now directly try out rules and screen descriptions, receiving immediate feedback to their work. Requiring them to specify requirements in a formal language also forces them to think through rules and procedures. If they wrote down analysis results as human-readable documents, the team would only discover the errors, inconsistencies, and lack of details during programming or even later. Moreover, the shift to using DSLs has impacted the amount of documentation analysts produce. Because they now implement some of the functionality themselves, the programs written in DSL become (formal and executable) documentation of the system's behavior. However, although this makes the representation clearer and more precise, client representatives who don't have a technical background can't easily understand these formal descriptions. Moreover, because such descriptions focus on the "what" or the "how" rather than the "why," there is still a need for human-readable documentation to give a nontechnical overview of the system's functionality and a rationale of key design decisions.

One (purported) advantage of using DSLs is the ability to involve domain experts in the system's design process.[3] This was also our initial intention when building CuE—for example, we specifically designed Burula with nontechnical

end users in mind. Our initial plan foresaw that our analysts would create the initial set of rules and, from that point forward, domain experts at customs would take over rule development. However, when we held training sessions for rule writing, we found that this plan wasn't practical. Users struggled with even the most basic concepts of computer programming (such as a need to express themselves in a very formal, constrained language). We found it more cost-effective to train our own business analysts (who have a more technical background than the end users) rather than training a larger pool of domain experts who didn't have any IT background. Fortunately, after our analysts wrote the verification rules, the domain experts were able to read and sometimes even modify them.

One practical issue that we encountered was that of tool support for our DSLs. By definition, DSLs aren't mainstream languages and therefore lack the high-end tools created for more popular options, such as Java. Because CuE contained several DSLs, we couldn't spend many resources on creating specialized tools for each language, so we added syntax highlighting support for a few common editors instead of integrated development environment-style tools with full support for autocompletion, real-time syntax error detection, integrated debugging, and so on. This frustrated some programmers who felt that writing DSL programs with a standard text editor wasn't as convenient as writing the same functionality in Java, although the latter might be longer and potentially less readable. Although tool support for visual DSLs based on the concepts of model-driven engineering has received significant attention,[6] this support is yet to be fully transposed to textual DSLs in the Java environment. Efforts in this direction[7] are under way, but at the time the CuE project started, their level of maturity wasn't sufficient to warrant their use in a production environment.

The most important DSL in CuE was Burula. The amount of effort spent on developing it (creating specifications, programming the compiler and runtime, integrating Burula into CuE subsystems, creating a toolset) was roughly four man-months, with the cost shared among six CuE subsystems. A moderately complex CuE subsystem has roughly 100 to 150 verification rules, most derived from EU specifications. Developing the 131 verification rules for the ECS subsystem took approximately 2.5 man-weeks, primarily spent learning Burula, reading and analyzing the data model and specifications on data constraints, and writing and testing the rules.

About the Author

Margus Freudenthal is a PhD student in the Department of Computer Science at the University of Tartu. He also works at Cybernetica AS as a researcher and software developer. His research interests include domain-specific languages and software reuse in the context of developing enterprise systems. Freudenthal has an MS in computer science from the Tallinn Technical University, Estonia. Contact him at margus@cyber.ee.

The most complex CuE subsystem is SAD, which manages import and export declarations and contains roughly 2,000 verification rules derived from EU legislation, national regulations, and Estonian Customs working procedures. During the first 36 months after the subsystem went into production, customs updated the verification rules in the production environment 56 times—roughly 1.5 modifications per month. Most of these updates were bug fixes, but changes in requirements (such as legislation or working procedures) caused 24 cases.

The DSL approach gave us positive results, with benefits clearly outweighing costs. Although we spent extra effort developing the DSLs, it helped us achieve a high level of reuse and a flexible, maintainable system. Based on our experience, I would suggest that the DSL approach is most beneficial in projects that

- have a complex, frequently changing business logic;
- are sufficiently large so that the team can use DSLs created in the project in several places; and
- have specialized developers and tools for efficiently building DSL support.

Our work on CuE's development continues, with goals for further research including performing more formal usability analysis to determine which language characteristics make more sense for nontechnical users and improving the existing level of tool support. We plan to pursue this by exploiting ongoing research in the field of model-driven engineering and adapting it to deal more effectively with textual DSLs and combinations of visual and textual DSLs.

Acknowledgments

Thanks to Arne Ansper, Madis Janson, and the rest of the CuE team for their fruitful collaboration.

Thanks to Marlon Dumas for his feedback on earlier drafts of this article. This research was supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS.

References

1. A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *SIGPLAN Notices*, vol. 35, no. 6, 2000, pp. 26–36.
2. D. Batory et al., "Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study," *ACM Trans. Software Eng. Methodology*, vol. 11, no. 2, 2002, pp. 191–214.
3. A. van Deursen and P. Klint, "Little Languages: Little Maintenance?" *J. Software Maintenance*, vol. 10, no. 2, 1998, pp. 75–92.
4. S. Chandra, B. Richards, and J. Larus, "Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols," *IEEE Trans. Software Eng.*, vol. 25, no. 3, 1999, pp. 317–333.
5. S. Thibault, R. Marlet, and C. Consel, "Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation," *IEEE Trans. Software Eng.*, vol. 25, no. 3, 1999, pp. 363–377.
6. J. Grundy et al., "Generating Domain-Specific Visual Language Editors from High-Level Tool Specifications," *Proc. 21st Int'l Conf. Automated Software Eng.* (ASE 06), IEEE CS Press, 2006, pp. 25–36.
7. F. Jouault, J. Bézivin, and I. Kurtev, "TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," *Proc. 5th Int'l Conf. Generative Programming and Component Eng.* (GPCE 06), ACM Press, 2006, pp. 249–254.