Developing a Domain-specific Language for Scheduling in the European Energy Sector

Stefan Sobernig[1], Mark Strembeck[1], and Andreas Beck[2]

[1] Institute of Information Systems, New Media Lab
Vienna University of Economics and Business (WU Vienna), Austria
firstname.lastname@wu.ac.at
[2] VERBUND Trading AG, Austria
andreas.beck@verbund.com

Abstract European electricity companies trade electric power across country and market boundaries. So called *schedules* are data sets that define the terms and conditions of such power trades. Different proprietary or standardized formats for schedules exist. However, due to a wide variety of different trading partners and power markets, a number of problems arise which complicate the standardized exchange of schedules. In this paper, we discuss a project that we conducted to develop a domain-specific language (DSL) for scheduling in a large Austrian electricity company running more than 140 power plants. The DSL is written in Ruby and provides a standardized programming model for specifying schedules, reduces code redundancy, and enables domain experts ("schedulers") to set up and to change market definitions autonomously.

Keywords: Domain-specific Language, DSL, Power Market, Power Trading, Scheduling, Industry Project, Europe

1 Introduction

The VERBUND AG[1] is an Austrian electricity company and one of the largest producers of electricity from hydropower in Europe. VERBUND AG has about 3.300 employees and is running more than 140 power plants in Austria and other European countries (125 of which are hydropower plants) to serve about one million private households and corporate customers. The VERBUND Trading AG (VTR)[2] is a subsidiary company of VERBUND AG. VTR is the operating unit for the optimization of the power plants, for international power trading, and for the scheduling process of the VERBUND group and its subsidiaries. VTR trades about 500 GWh of electrical power on a daily basis (500 GWh correspond to an annual electricity consumption of some 120 000 households). In the VTR context, *schedules* are structured data sets which contain the technical details

[1] http://www.verbund.com/cc/en/
[2] http://www.verbund.com/cc/en/about-us/our-business-divisions/trading

about the terms, the conditions, and the volumes of the power deals made within or across 21 European power markets in 18 countries.

A transmission system operator (TSO) is a company that runs an infrastructure (the power grid) for transporting (electrical) energy. The scheduling process ensures the transfer of schedules to the TSO concerning the amount of power traded in each of the TSO's markets. The liberalization of the energy market (which occurred in 2001 in Austria) requires that the exchange of the schedules between the TSO and its trading partners be carried out in a standardized way so as to guarantee the quick and automated processing of schedules and to ensure the maintenance of a stable power grid.

However, due to the large number of different energy markets and trading partners, VTR faces a number of problems with respect to a standardized exchange of schedules. The first problem is the heterogeneous set of applications and scripts that VTR currently uses to carry out its scheduling process. The legacy system that VTR employs to support the scheduling process has been in use since 2007. The system is based on a number of different Microsoft Excel workbooks and embedded spreadsheet applications (Visual Basic for Applications, VBA, macros). The bulk of the source code used for data retrieval, calculations, and format-building logic is the same in every workbook (code clones). There are, however, differences between the workbooks arising from the implementation of the local *market rules* (rules that are specific for a given power market) in each workbook. Therefore, the maintenance of the code requires substantial effort because, in order to maintain a consistent code basis, every change in one workbook must be incorporated into every other workbook. VTR reports on having spent an average of 30 person-days per year since 2007 on maintaining and further-developing the existing scheduling-support system. The second problem is to develop the technical knowledge needed to make changes to the workbooks throughout the organization and to render the scheduling system adaptable by non-technical domain experts ("schedulers"). At the same time, any scheduler wishing to make changes must comply with company requirements and work within the existing system landscape.

In this context, we developed a scheduling system based on a domain-specific language (DSL) to address the above problems. The project started in January 2011 and has evolved over 2.5 years. For this project, we applied an extraction-based DSL development style [15] to systematically define a DSL for scheduling in the energy sector based on the existing scheduling system. In the remainder, we report on this development project and its exploratory evaluation by first providing some background on the scheduling domain (see Section 2). We then document the DSL design and implementation in Section 3. Based on an early case-study evaluation reported on in Section 4, we review the achieved benefits of the DSL-based system refactoring (see Section 5) and the lessons learned from applying an extraction-based DSL development style (see Section 6).

2 Background: Scheduling Power Deals

During the scheduling process, VTR must handle different representation formats of schedules which detail the amounts of power VERBUND AG produces, the amounts of power it imports, the amounts it exports (deliveries that cross market boundaries), and the amounts of power it trades in internal areas (deliveries within a market). Furthermore, VTR must deliver those schedules to various market-specific recipients, including TSOs. Each market has its own schedule. Runtime occurrences of schedules are referred to as *schedule messages* (messages, hereafter), with each message having its particular format, covering a particular scope, and relating to a particular mode of transmission. The most common message formats are ESS (ETSO Scheduling System [5,6,7]) and KISS (Keep It Small and Simple [4]). ESS is a special-purpose XML-based data format, while KISS is based on Microsoft Excel. Moreover, for some markets VTR must provide proprietary (mostly Excel based) formats. The scope of a particular schedule can cover market-internal power deliveries, market-external power deliveries, or both. A schedule can be transmitted either via e-mail, FTP, Web applications, Web services, or any combination of these. Figure 1 illustrates this configuration space of schedules.

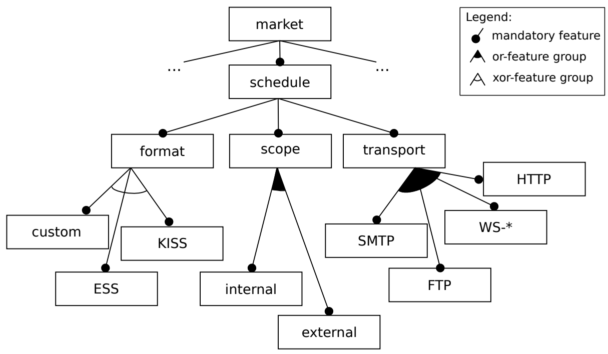


Figure 1: Excerpt from the Scheduling Domain Model, depicted as Feature Diagram [2].

There are some principles that every schedule follows. Every delivery or receipt relationship is called a *time series*. Every time series is identified by a set of elements. Every schedule has the following elements (although the name for a particular element may differ, depending upon the format used for the particular schedule): In Area, Out Area, In Party, Out Party, and the Capacity Agreement ID. There are other elements that may elaborate upon the time series but those elements are not part of the distinct identification of a schedule. Each time series also contains the delivery quantities of the traded power. A particular time series covers either the amount of electrical power delivered to one particular customer or the amount of power received from a supplier in one control area.

The In and Out Area define the direction of the energy flow. The Out Area is the market area the energy comes from and the In Area is the market area to which the energy goes. For the identification of the areas, standardized codes are used. Time series covering deliveries within a market area (i.e., an internal delivery) have the same In and Out Area code, while time series covering deliveries across two market areas (i.e., external or cross border delivery) bear the respective area code for each market. The concept of the trading border is similar to geographical borders, except that a trading border only exists where there are power lines in place that link two markets. There are, for example, four different market areas within Germany but only three have a border with and power lines that link to Austria. As a result, there are trading borders between three of the German markets and the Austrian market. Another example is the border between Austria and Slovakia. There is a geographical border, but, because there are no power lines connecting the two countries, there is no trading border between them.

The In and Out Party elements define separately the party which delivers and the party which receives the energy. The Capacity Agreement ID is only applied to cross border deals and identifies capacity rights. The term “capacity right” refers to the right to export or import power across a specified trading border. On limited borders, i.e., borders with limited power transmission capacity, one of the parties must have a capacity for the import or export of power. Although the required market-specific formats may dictate differing usage of certain elements, VTR defines a core data schema and value ranges across markets and across different formats. In some markets, however, VTR employs proprietary message formats which are fully customized. Additionally, the message exchange protocols used to transmit time series can vary as well. The usual ways used to transmit such series are SMTP (with attachments) and HTTP POST (via a web application).

The number of recipients to which VTR sends the schedule messages also varies from market to market. The schedules for these markets contain information about the power flows both across and within market borders. A mandatory recipient of any schedule is the TSO but recipients can include other official power market parties, such as power or energy exchanges like the EPEX (European Power Exchange) in Germany and France or market makers in general like Borzen in Slovenia, or OTE in the Czech Republic. In Austria, there are two recipients. One is the TSO (the Austrian Power Grid AG) which receives only data concerning power flows across the country’s borders and the other is the APCS (Austrian Power Clearing and Settlement AG), to which all the transaction data inside the Austrian market are sent.

The daily transactions result in a high number of schedules. On an average day, the number of schedules may go as high as about 200. The types of schedules that VTR generates include long-term, day-ahead, intra-day, and post-scheduling schedules. Long-term schedules cover the time frame before D-1, where D stands for the delivery day. That is, if the delivery day is June 14, every schedule sent before June 13 is considered to be long term. The valid time frame for a specific market can be found in the local market rules. The time frame for day-ahead usually begins the day before delivery and ends after the final schedules for the delivery day are sent, which usually is on D-1 at about 14:30. In our example that would be June 13, 14:30. Again, the valid time frame for a specific market can be found in the local market rules. Intra-day schedules cover the time frame between the start of intra-day (which is usually shortly after the end of day-ahead) and the end of the delivery day. Post-scheduling schedules cover the defined time frame after delivery.

At VTR, long-term and day-ahead schedules are handled by a group of 4 schedulers. Intra-day scheduling is handled by an intra-day trading team, which handles both intra-day scheduling and controls the power plants. Post scheduling is again handled by the schedulers group. For verifying data quality, the standing data (e.g., market definitions) and the transaction data (as retrieved by executing data queries in the trading system) entering a schedule message are subjected to review processes. The standing data must undergo a double-check by a second scheduler to ensure their correctness. The transaction data at VTR are double-checked upon processing by the power trader and the back-office staff.

Table 1: Key Figures for the Legacy Scheduling System

Number of implemented power markets	18
Number of Excel workbooks	14
Avg. SLOC[3] per workbook	1 500
Avg. market-specific SLOC[3] per workbook	50

3 A DSL for Scheduling

In addition to implementing the domain of scheduling as analyzed in Section 2, the DSL-based scheduling system sets out to address a number of objectives. The following four goals resulted from the actual difficulties with the existing scheduling system as experienced by administrator and schedulers at VTR.

Minimize Code Clones. In the existing scheduling system, 14 Excel workbooks generate different schedule types for 18 of the 21 markets (see Table 1). For the remaining three markets, third-party schedule generators are used which are not maintained by VTR. On average, there are about 1 500 source lines of code (SLOC[3]; mainly VBA code) in each workbook, for a total of approximately 21 000 SLOC in the 14 workbooks. The average number of market-specific SLOC per market is approximately 50, summing up to 700 lines in 14 workbooks. Therefore, when comparing the code bases of the workbooks, the workbooks share approximately 97% of their code bases. Only the remaining 3% are code

[3] The source lines of code (SLOC) were measured using `cloc` [3].

fragments specific to single markets. Specialization involves local market rules relating to message generation, such as those used in the mapping of codes (e.g., market-area codes). Such rules usually follow a certain default rule that makes it easy for human beings to read and to understand them. In some cases, however, certain values must differ from default rules, such as crossing in and crossing out of a control area for cross-border energy deliveries. Table 2 exemplifies configuration data specific to the power market of the German company "Rheinisch-Westfälische Elektrizitätswerks Aktiengesellschaft" (RWE).

Table 2: Schedule Configuration Specific to the RWE Market

Configuration point	**Configuration value(s)**
Format	ESS V2R3
Borders	EON, ENBW, LU, AT, FR, CH
SenderIdentification	13XVERBUND1234-P
SenderRole	A01
ReceiverIdentification	10XDE-RWENET—W
ReceiverRole	A04

The numerous code clones make the workbooks hard to maintain over time. Propagating the latest version of the code to every workbook is tedious and time-consuming. For example, one basic step within the scheduling process is the use of Business Objects reports provided by the SAP business intelligence software [12] as part of the power-trading system. SAP Business Objects offers a COM-based API for generating such reports. Whenever there are SAP vendor upgrades, there are API changes affecting any client application such as the workbooks. As a result, one must examine every workbook to reflect these API changes. In the majority of shared workbook code, a second source of redundancy are recurring configuration data, common to all or subsets of markets. For example, the various input and output identifiers (e.g., file names, output directory names for messages and temporary files) follow one naming convention.

Establish Participatory Maintainability. There are two main participant roles in the existing scheduling process. On the one hand, there is the scheduler as the non-technical domain expert, and, on the other hand, there is the administrator responsible for setting up new markets. The scheduler, as the domain expert, has a deep knowledge of the scheduling process, of the message formats used, of the local market rules, and of the delivery modes for messages. The administrator, as the primary workbook developer, knows the programming logic and the design of the workbook code. In the legacy system, almost everything from the GUI logic that governs the configuration to the business logic has been included in the monolithic workbook code. This means, of course, that the administrator is the only one who can make even the smallest changes or amendments to the code. The objective, therefore, is to develop a system that will allow the schedulers to participate in implementing new markets, with new rules, only requiring action by an administrator for non-routine tasks (e.g., code changes due to new interface versions of SAP Business Objects).

Cover Standard and Custom Message Formats. For 18 of the 21 power markets mentioned above, VTR uses standardized message formats, such as the Excel-based KISS or the XML-based ESS formats. However, VTR must be able to derive custom, market-local message formats from standard formats to accommodate market rules deviating from the standards. The concrete format for a market is usually a deviation from a standard format and may be defined by the market operator or energy controlling authority. Occasionally, even the rules set by a TSO may differ from the standard or officially advertised rules.

Integrate Refactored System into the Existing System Landscape. To keep changes to the existing technical landscape at VTR to a minimum, the DSL development should avoid the introduction of new software components. Software components already in use in the existing scheduling process are Excel, Business Objects, as well as VBA and VB.NET as frontend languages to .NET as runtime platform.

Provide Uniform, but Variable Graphical User Interfaces (GUI). Each workbook provides a unique UI form to the scheduler for entering configuration data, such as the delivery date, delivery message, or the message format to use. These scheduler forms tend to be narrowly focused and tailored to meet the exact needs of each particular market. This is, however, not a flaw in the design of the forms. Rather, the narrowness is simply a result of the extreme tailoring towards scheduling needs in a particular market. Nevertheless, the deviating form designs affect a scheduler's ease to move between the workbooks negatively, because they require the user to interpret and to understand every different user form. The goal, therefore, was to develop a uniform user interface that would contain cross-market functionality to run schedules, but which would allow the scheduler either to activate or to deactivate GUI parts deemed necessary or unnecessary for a given market.

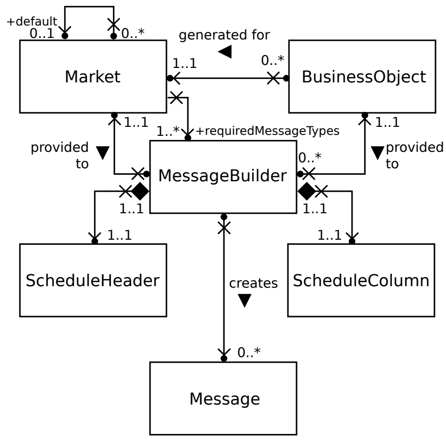
Given these requirements and restrictions, it was decided to develop an embedded DSL using Ruby as its host-language infrastructure. An embedded DSL meant a minimal and non-invasive addition to the existing system landscape without the need for adding software components, for example, for parsing and integrating an external DSL. Besides, an embedded DSL provides for seamless integration with the existing runtime infrastructure such as the SAP business intelligence software (see, e.g., [15,19]). The dominant interface for domain users (schedulers) was to remain a revised, form-based GUI with the embedded DSL serving as an alternative backend syntax for maintenance tasks, rather than as a complete replacement. Ruby was adopted because of its suitability for developing embedded DSLs (see, e.g., [8]) and its availability as IronRuby for .NET [9], including IDE support by Microsoft, which is the required development platform at VERBUND Trading AG.

3.1 Language Model and Concrete Syntax

The language model and the concrete-syntax style were extracted from reviewing the documentation, the code base, and auxiliary documents of the existing software system [15]. The reviews were performed by the DSL design team consisting of the three authors. The main domain abstractions, which constitute the core language model, were identified by studying, first, the Excel workbooks because they host the existing VBA source code and the existing market definitions. Second, Business Objects reports were investigated for the trading data they provide and for the business logic represented by data queries. Third, there are the standards documents defining the schedule-message formats, including markup-schema definitions such as the ETSO Scheduling System formats [5,6,7].

Abstract Syntax. Figure 2a shows the conceptual language model of the DSL. The key abstraction is the `Message` which represents a schedule in a specific message format. A `Market` models a power market and records important market-specific data such as the market symbol, border codes, and the allowed message formats. `Market` can refer to one direct `default Market` whose configuration data is inherited if not redefined. A `MessageBuilder` implements a generator for a specific message format which specifies a `Message` in terms of construction rules for a `ScheduleHeader` and a `ScheduleColumn`. The attributes of `ScheduleHeader` enter the message headers as required by the message format. A `ScheduleColumn` represents the different time series which form the body of a schedule message. The construction rules typically take the form of mappings and, if needed, transformations between message elements (e.g., as specified by the KISS and ESS message formats) and the elements of a given `Market` definition. Depending on the message representation (e.g., XML), the construction rules may also specify the representation creation. For writing XML markup, our DSL integrates with the Builder library available for Ruby [18].

Concrete Syntax. In the existing, workbook-based scheduling system, the configuration data for markets and schedule messages are maintained in a rows-and-columns spreadsheet format with assigning certain rows, columns, or individual cells the role of meta-data stores identifying the meta-data type using a text label. Both roles, administrators and schedulers, performed their tasks using this tabulated syntax. To meet the requirements of a uniform GUI for domain users (i.e., schedulers) and to separate the user interfaces between administrators and schedulers (see requirements above), a textual concrete-syntax for administrators as primary users of the scheduling DSL was devised. Listing 2b shows the `Market` definition for the control area RWE and an exemplary `MessageBuilder` definition of the KISS message format, showing the assignment of a market-name symbol, the six transmission network borders relevant for this market,



(a) Core Language Model

```
Market :RWE do
  @symbol = :RWE
  @borders = [:EON,:ENBW,:LU,:APG,
             :FR,:CH]
  @requiredMessageTypes = [:KISS]
  MessageBuilder :KISS do
    schedule do
      @OutArea = ""
      @InArea = ""
      @OutParty = ""
      @InParty = ""
      @Oasis = ""
      @InitialTerm = ""
    end
  end
end
```

(b) Concrete Syntax Example

Figure 2: Scheduling DSL

and the message format to use for this market: KISS. Note that for KISS only, the variables are initialized using empty strings.

As an embedded DSL, the textual-concrete syntax leverages and integrates with the concrete syntax of the Ruby host language. To reflect the use of the DSL for configuration programming of markets and message builders, especially mapping and construction rules, a single assignment form is promoted. Assignments establish correspondences between elements of market definitions and message formats, on the one hand, as well as between market attributes and configuration values, on the other hand. This syntax style is realized using the principles of object scoping and nested closures [8].

Constraints. There are constraints applying to a `Market` and a `MessageBuilder`. The `Market` must have a unique name accompanied by a unique symbol or abbreviation (e.g., “RWE”). This pair represents the market or control area for which this definition stands. Furthermore, the possible borders for power import and export have to be stated and the message types applicable to this market. Each message format has then to be represented by a `MessageBuilder` referenced by a `Market`. In addition, there are specific constraints on the data representations of trading data. For example, value constraints on time stamps are set by the ESS family of standards (see, e.g., [5]).

Structural Semantics. The inheritance semantics between a `Market` and its default `Market` are those of concatenation-based prototypical inheritance [16]. This allows for factoring out common configuration data into single and reusable `Market` definitions. This way, creating incremental variants of single market definitions upon changed market requirements eases maintenance. Implementing this refinement scheme is facilitated by the use of nested closures (see below).

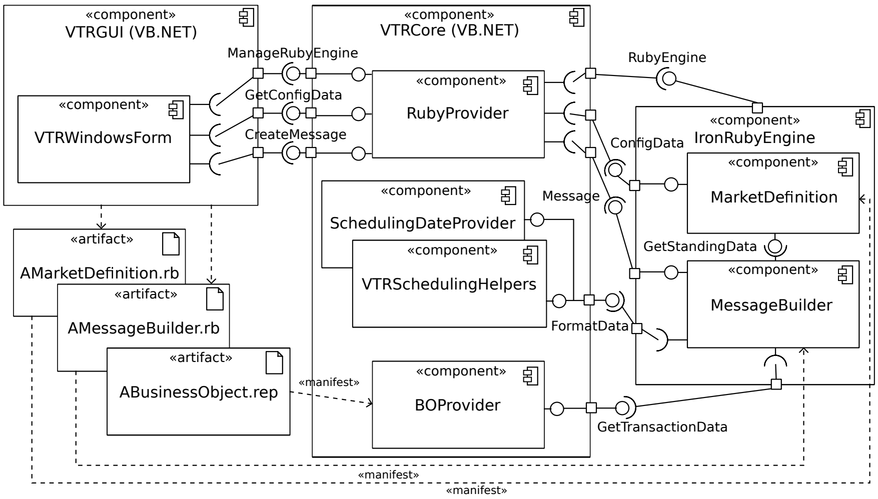


Figure 3: An Architectural Overview of the DSL-Based Scheduling System

3.2 Implementation

The realized architecture is built from three interacting components: a GUI component (`VTRGUI`), a managed assembly (`VTRCore`), and the `IronRubyEngine` (see Figure 3). The GUI and the managed assembly are implemented using .NET 4.0 and VB.NET as frontend language. The retrieval process for transaction data uses Microsoft Excel 2010 and SAP Business Objects. The Ruby engine is provided by IronRuby 1.1.3, a Ruby implementation targeting the Microsoft Common and Dynamic Language Runtimes (CLR, DLR) and widely complying with MRI-Ruby 1.9.2.

The overall workflow of creating a `Message` is controlled by the GUI component in terms of a wizard. Once the scheduler has selected a market definition (`AMarketDefinition.rb`) and a message-builder definition (`AMessageBuilder.rb`), the GUI sets up a Ruby evaluation context using the `RubyProvider` component. Based on the standing data for the selected market, the `VTRCore` retrieves the transaction data in terms of a Business Objects report using the `BOProvider` and provides market-specific configuration data to update the message-creation wizard (e.g., available control areas). The scheduler then completes the message-configuration step and has the selected `MessageBuilder` create the final `Message`. The `MessageBuilder` formats the transaction data using helpers such as the `SchedulingDateProvider` provided by the `VTRCore`.

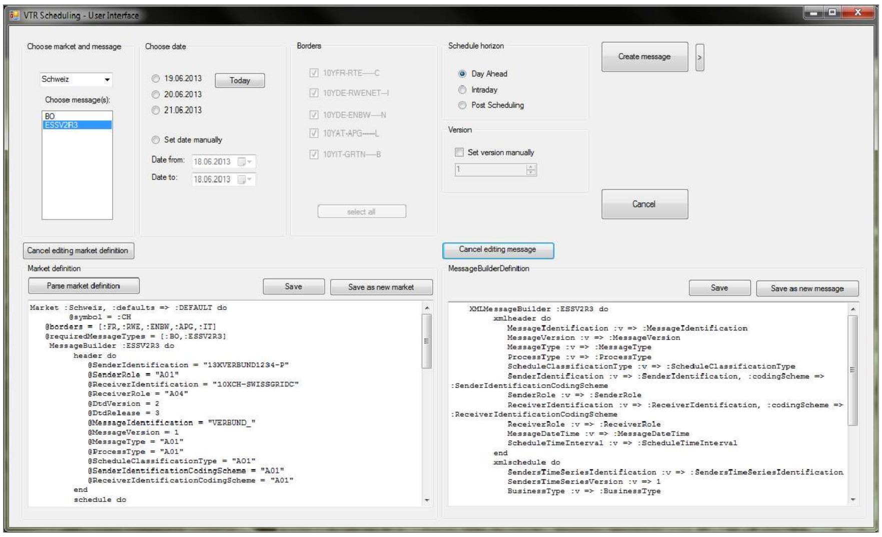


Figure 4: The Controlling GUI of the DSL-Based Scheduling System.

The language model is implemented by a Ruby class collaboration by mapping the entities in Figure 2a to Ruby classes. Ruby provides built-in language mechanisms to realize nested closures, object scoping, and instance evaluation [8]; the techniques used to implement the structural semantics and the concrete-syntax style outlined before. A Ruby block (also called a Ruby Proc or a closure) is a group of executable statements defined in the environment of the caller and is passed unevaluated to the called method as an implicit argument. The called method may then execute the block zero or more times, supplying the needed arguments for each block evaluation. For example, a context variable storing a reference to a `Market` can be provided to the `MessageBuilder` when processing the block which stores the message production rules. To populate a `Market` or to construct a `Message` from a `MessageBuilder`, the market-definition scripts and the message-builder scripts, which are implemented as blocks, are evaluated in the scope of instance objects of `Market` and `MarketBuilder`. This principle is referred to as object scoping [8]. In addition, this allows for implementing the concrete syntax of `Market` and `MessageBuilder` as an expression builder [8]: For this reason, each `Market` and `MessageBuilder` keyword (see, e.g., `schedule` in Listing 2b) is implemented as a Ruby method. By limiting evaluation to defined accessors, methods, and classes, there is scaffolding of schedulers to only use this pre-defined vocabulary. Populating a `Market` and creating a `Message` are controlled by the `VTRCore` component, by instrumenting Ruby entities in their .NET representation using cross-language method invocations [14].

The Ruby-based DSL implementation is written in 500 SLOC[3], the VB.NET-managed `VTRCore` component has a code base of 1 100 SLOC, and the GUI amounts to 800 SLOC in VB.NET. Figure 4 shows the GUI wizard having loaded a definition of the Swiss market and the ESS 2.3 message builder. The wizard provides views for both the administrator and the scheduler roles, with the administrator being able to manipulate the market definitions and message-builder definitions directly.

4 DSL Evaluation

To assess whether the DSL-based scheduling system meets the previously defined requirements (see Section 3), we designed a case study [11]. In the following, we summarize the case study objectives, the real-world setting to be studied (the case), important details of data collection (collection techniques, actors), and key observations. A complete account on the case study and on supplementary evaluation steps (e.g., scenario testing of the prototype) is given in [1].

As for *case selection*, we picked the task of defining a new power market to generate schedules for this market. The power market to be implemented was not only required to be representative, but it should also involve complex and large-sized schedules and time series. In addition, it should cause a high frequency of message generation in a trading time window and a comparatively high number of trading partners as message recipients. The selected case dealt with the generation of schedule messages for the German market area RWE. This is the most important market area for VTR in Central Europe in terms of the traded energy amounts. There are 350 active traders in that control area and the schedules VTR sends to the TSO (Amprion) of that control area contain more than 100 time series, each one identifying either a delivery or a receipt of energy to or from one counterparty.

The *case objectives* were twofold: First, the DSL-based and the legacy scheduling systems were to be exercised by implementing the RWE market. The two procedures of setting up a new market were to be performed by a VTR scheduler, sufficiently proficient in using both scheduling systems. Second, the DSL-based prototype was to be evaluated against the critical timing requirements on generating schedules for the RWE market. Schedule generation and delivery are time-critical in the range of 1 or 2 minutes in certain markets including RWE. This is because energy trading happens in fixed time boxes (e.g., 15 minutes after the hour) and price increases tend to grow towards the end of trading windows. Trading, however, is stopped effectively before the end of a time box to create and to deliver the schedule messages reliably for completing the transaction. To optimize an intra-day trading portfolio, the energy seller seeks to minimize schedule-handling times to extend the effective trading time.

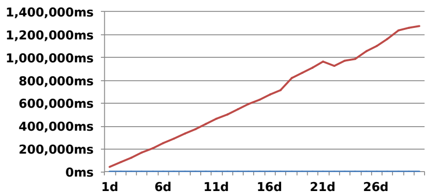
The *case work* was performed by the third author who, as a VTR energy manager, can take both the roles of the administrator and the scheduler for the legacy and the DSL-based scheduling system. Performing the case study involved preparatory steps identical for each system. These steps included gathering the market rules for the RWE market area and interfacing with the power-trading system to obtain the trading data in terms of Business Objects reports. The main task was then to implement the RWE market rules, once using the scheduling

DSL and once using an Excel workbook. For this last step, we recorded the working time needed, collected the resulting code artifacts (VBA macros, DSL-based market definition), and monitored for runtime data to learn about the time and space efficiency of the DSL-based scheduling system, especially when generating schedules.

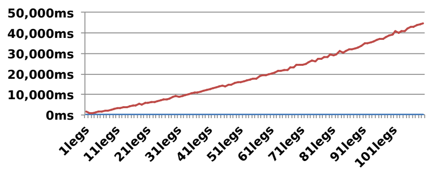
Table 3: A Scheduler's Work Station Configuration at VTR.

Processor	Intel U9600 @ 1.6 GHz
Memory (RAM)	3.00 GB
Hard Drive	60 GB SSD
Operating System	MS Windows 7 Enterprise x64
Office suite	MS Office 2010

The key observation was that the effective market-definition time using the DSL amounted to approximately 10 minutes, while the definition process in the legacy system required an entire, 8-hours working day (i.e., one person-day). This substantial effort escalation in the legacy system was due to the tedious and time-consuming task of screening existing Excel workbooks for code fragments to be reused directly or, mostly, in a modified form.



(a) 110 Time Series per Schedule/Day, 1-30 Days.



(b) 1-110 Time Series per Schedule/Day, 1 Day.

Figure 5: Schedule Generation Times of the DSL-Based Scheduling System

In light of the strict timing requirements, we ran time-efficiency measurements. Time efficiency was assessed by measuring the elapsed execution time between the start and the end of the schedule-creation process in milliseconds on a scheduler's typical work station (see Table 3). We devised different data sets as representative workloads for distinct scenarios: a fixed-size schedule containing 110 time series ("legs") of one market for 30 trading days and a schedule growing by one time series per iteration for one trading day. Overall, we found linear growth patterns for these workloads (see Figure 5a and Figure 5b). For large-sized schedules (110 time series per schedule), average processing times of 43 seconds were measured. This compares with approximately 1.5 minutes for similarly sized schedules in the legacy system. For smaller sized, growing schedules (one up to 110 times series per schedule), the average processing time was approximately 345 milliseconds.

5 Achieved Benefits

Reduction of Code Redundancy. Where the old system required separate Excel workbooks for each market implementation, the DSL only uses the market-specific configuration artifacts called *market definitions* expressed in a Ruby-based embedded DSL. In addition, general configuration settings, valid for several markets, can be placed once into reusable market definitions used together with specializations to generate a market-specific schedule. Different message formats (KISS, ESS) are defined in a second set of DSL scripts referred to as message builders. Again, message-format details only need to be maintained in one central location rather than in separate workbooks. This also applies to defining new, custom message formats.

Standardized Interface. The GUI implementation has been centralized and unified as well. Initially, the GUI component reads the available markets from the market definition, identifies the market-required message formats, and automatically updates that information in the GUI (e.g., by providing market-specific drop-down lists and check boxes). In a next step, the GUI provides market-specific configuration steps to the scheduler, such as the different trading borders or trading times. This runtime adaptation allows for the GUI code to be reused across market implementations. On top, the GUI is used to provide a uniform representation to the Business Objects reports, as basis for handling transaction data in a standardized and a consistent manner across markets.

Scheduler Participation. The DSL-based scheduling system renders selected internals of the scheduling accessible to and adaptable by the non-technical domain experts, the schedulers. That is, the scheduler can be trained with little effort to perform small and anticipated changes to market definitions and message builders on duty, based on a syntax reflecting her domain terminology and without requiring deep knowledge on the underlying software execution platform (.NET, Ruby). The revised GUI providing a consistent view on markets and transaction data facilitates collaborative tasks and context switching, such as in peer reviews of transaction data between schedulers (see Section 2).

Whereas the systematic design process was primarily driven by artifact reviews (see Section 3.1), a late prototype of the DSL-based scheduling system was used to set two schedulers, the target audience of the DSL, in the future situation of working with the prototype. In separate ad hoc sessions, each scheduler was guided through the schedule-generation process by the third author, a former scheduler at VTR himself. Immediate feedback was collected orally, in particular, feedback on the GUI, on the DSL-based procedure for defining a new market, and on whether the generation times were acceptable. Defining markets using the DSL was judged intuitive by the two schedulers. Having the GUI adapted immediately in response to changes in market definitions was deemed useful. This positive feedback did not require any modifications to the actual language design, that is, the abstract syntax, the concrete syntax, the constraints, and the structural semantics (see Section 3.1).

Improved Concern Separation. The DSL-based scheduling system cleanly separates between the concerns of defining/maintaining a market and defining/-maintaining a message format. For example, a scheduler can implement a new market and the pre-defined message formats (e.g., ESS 2.3 and KISS) can be applied to schedules for this market directly. Conversely, when implementing a new message format (e.g., another ESS revision), this format becomes available to the base of market definitions. In the legacy system, such additions or changes required modifications in all affected market implementations (workbooks).

6 Discussion

The development of this scheduling DSL did not occur in a vacuum. Rather, it was an enhancement of an existing system. The existing scheduling system presented us with a number of benefits and liabilities during the DSL development process. One benefit from working with the existing system was that we could derive the domain abstractions (e.g., market, schedule, message) from that system [15,19]. A second benefit of investigating the existing system was that this system clearly defined the scope of the DSL, as well as functional and non-functional requirements on the DSL [19]. For example, in the evaluation phase, we established a baseline of execution timings and working times using the legacy system. An existing system, however, poses the potential liability that relying upon the existing domain abstractions could hinder the critical review and adoption of revised domain concepts [19]. As a result, the extracted DSL could be limited in its expressiveness. We addressed this risk by conducting a domain analysis beyond the narrow boundaries of the existing scheduling system, by including standards documents available for the scheduling domain.

As for the concrete-syntax style of the embedded scheduling DSL, a textual concrete syntax and a graphical frontend syntax were adopted [15]. Under this approach, the basic configuration data are stored as text, and the representation of such data for the user is done by means of a GUI. The textual syntax representations of market and message-builder definitions are interpreted and rendered, especially for the scheduler role. The choice of using a textual concrete syntax for DSL development has a number of benefits. With this syntax, market and message-builder configurations can be specified in a compact manner and existing editors for Ruby can be reused [19]. Furthermore, a textual concrete syntax in support of a graphical frontend helps separate different working tasks for individual domain users. For the repetitive and routine task of generating schedule messages, the visual frontend allows for acquiring a quick overview of standing and transaction data. The non-routine task of modifying or creating market and message-builder definitions can be achieved in a compact textual form. A drawback of a mixed textual and graphical syntax is the need for scheduler awareness of subtle interdependencies between the two syntactic representations of domain concepts.

7 Conclusion

Documented and systematically collected empirical evidence on the alleged benefits of DSLs such as an improved maintainability [17] in an industry setting is rare (see, e.g., [13]). In this paper, we report on a successful development and deployment project of an embedded DSL for the VERBUND Trading AG (VTR), the subsidiary company responsible for power trading of the large-scale Austrian electricity company VERBUND AG. The project was carried out in a period of 2.5 years and included phases of domain analysis, DSL design and implementation, and an empirical evaluation based on a case study design and auxiliary software measurement. The DSL-based scheduling system is being actively used as a training tool for schedulers and as a backup scheduling system. VTR is planning to adopt the DSL-based system as a full replacement of the legacy system.

This project report shows that a DSL-based system refactoring can provide benefits in terms of reduced code redundancy for an improved maintainability of a code base. By enabling non-technical domain experts (schedulers) to participate in maintaining DSL-based system artifacts (e.g., market definitions, message builders), maintenance times can be reduced substantially. Finally, the project demonstrates that developing a DSL by extracting the DSL elements (e.g., its language model) from an existing system [15] represents a viable software-refactoring strategy [10] in otherwise rigid enterprise system landscapes. In follow-up work, we will perform more comprehensive and confirmatory empirical evaluations (e.g., domain-expert interviews, controlled experiments with domain-expert subjects) to reflect on the daily working routine based on the new DSL-based scheduling system at VERBUND Trading AG.

References

1. Beck, A.: Development of a domain specific language for scheduling in the energy sector. Master thesis, Institute of Information Systems and New Media, Vienna University of Economics and Business (August 2013)
2. Czarnecki, K., Eisenecker, U.W.: Generative Programming — Methods, Tools, and Applications. 6th edn. Addison-Wesley Longman Publishing Co., Inc. (2000)
3. Danial, A.: Count lines of code. URL: `http://cloc.sourceforge.net/`, Last accessed: 2013-05-02 (2013)
4. Electric System Operator: Schedule management (KISS). URL: `http://www.tso.bg/default.aspx/schedule-management/en`, Last accessed: 2013-05-02 (April 2012)

5. European Network of Transmission System Operators for Electricity (ENTSO-E): Scheduling system implementation guide 2.3. URL: `https://www.entsoe.eu/fileadmin/user_upload/edi/library/schedulev2r3/documentation/ess-guide-v2r3.pdf`, Last accessed: 2013-05-02 (April 2003)
6. European Network of Transmission System Operators for Electricity (ENTSO-E): Scheduling system implementation guide 3.1. URL: `https://www.entsoe.eu/fileadmin/user_upload/edi/library/schedulev3r1/documentation/ess-guide-v3r1.pdf`, Last accessed: 2013-05-02 (June 2007)
7. European Network of Transmission System Operators for Electricity (ENTSO-E): Scheduling system implementation guide 3.3. URL: `https://www.entsoe.eu/fileadmin/user_upload/edi/library/schedulev3r3/documentation/ess-guide-v3r3.pdf`, Last accessed: 2013-05-02 (April 2009)
8. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley (2010)
9. IronRuby: IronRuby – the Ruby programming language for the .NET framework. URL: URL `http://www.ironruby.net/`, Last accessed: 2013-05-02 (2012)
10. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng **30**(2) (2004) 126–139
11. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Eng. **14**(2) (April 2009) 131–164
12. SAP AG: SAP Business Objects. URL: `http://www.sap.com/germany/solutions/sapbusinessobjects/large/business-intelligence/data-exploration/index.epx`, Last accessed: 2013-05-02 (2012)
13. Sobernig, S., Gaubatz, P., Strembeck, M., Zdun, U.: Comparing complexity of API designs: An exploratory experiment on DSL-based framework integration. In: Proc. 10th Int. Conf. Generative Programming and Component Eng. (GPCE'11), ACM (2011) 157–166
14. Sobernig, S., Zdun, U.: Evaluating Java runtime reflection for implementing cross-language method invocations. In: Proc. 8th Int. Conf. Principles and Practice of Programming in Java (PPPJ'10), ACM (2010) 139–147
15. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. SP&E **39**(15) (October 2009)
16. Taivalsaari, A.: On the notion of inheritance. ACM Comput. Surv. **28**(3) (1996) 438–479
17. Van Deursen, A., Klint, P.: Little languages: little maintenance? J. Softw. Maint. Evol.: Res. Pract. **10**(2) (1998) 75–92
18. Weirich, J.: Builder. URL: `http://builder.rubyforge.org/`, Last accessed: 2013-06-15 (2013)
19. Zdun, U., Strembeck, M.: Reusable architectural decisions for DSL design: Foundational decisions in DSL projects. In: Proc. 14th Annual European Conf. Pattern Languages of Programming (EuroPLoP'09). Volume 566 of CEUR Workshop Proceedings. (2009)