Adding Spreadsheets to the MDE Toolkit

Mārtiņš Francis, Dimitrios S. Kolovos,
Nicholas Matragkas, and Richard F. Paige

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK.
{mf550, dimitris.kolovos,
nicholas.matragkas, richard.paige}@york.ac.uk

Abstract. Spreadsheets are widely used to support software development activities. They have been used to collect requirements and software defects, to capture traceability information between requirements and test cases, and in general, to fill in gaps that are not covered satisfactorily by more specialised tools. Despite their widespread use, spreadsheets have received little attention from researchers in the field of Model Driven Engineering. In this paper, we argue for the usefulness of model management support for querying and modifying spreadsheets, we identify the conceptual gap between contemporary model management languages and spreadsheets, and we propose an approach for bridging it. We present a prototype that builds atop the Epsilon and Google Drive platforms and we evaluate the proposed approach through a case study that involves validating and transforming software requirements captured using spreadsheets.

1 Introduction

Spreadsheets are arguably one of the most versatile and ubiquitous tools in the software world. In software development, spreadsheets are often used to collect requirements [1] and software defects, to capture traceability information between requirements and test cases, and in general to fill in gaps not covered by other components of typical engineering tool-chains. Although one can reasonably argue that using spreadsheets instead of more sophisticated task-specific tools is more often than not a sub-optimal choice, their use in practice is too widespread to ignore. Despite their popularity among practitioners, spreadsheets have received little attention from researchers in the field of Model-Driven Engineering (MDE).

In this paper we propose an approach for treating spreadsheets as *first-class models* in MDE processes, by providing support for seamlessly integrating them in MDE workflows alongside traditional models (e.g. EMF-based models). Our approach enables engineers to perform a range of model management operations on spreadsheets including model (cross-) validation, model-to-model transformation and code generation. We evaluate the proposed approach on a case study in which requirements captured in a spreadsheet are validated for their correctness and used to generate requirements graphs. By doing so, we demonstrate that it is feasible, practical and beneficial to treat spreadsheets and (metamodel-based) models uniformly using the same set of tools.

The remainder of the paper is organised as follows. Section 2 provides a discussion on the background and motivation of this work. Section 3 discusses the challenges involved in bridging spreadsheets and contemporary OCL-based model management languages, and proposes a language-independent solution to the problem. Section 4 illustrates a prototype that implements the proposed solutions and enables the model management languages of the Epsilon platform to query and modify Google Spreadsheets; this in turn enables a range of model management operations (including transformations) to be carried out. Section 5 evaluates the usefulness of the proposed approach using a comprehensive case study, Section 6 provides an overview of related work, and Section 7 concludes the paper and provides directions for further work.

2 Background and Motivation

MDE elevates models to first-class artefacts of the software development process, and proposes the use of automated model management (code generation, model transformation etc.) to try to enhance both the productivity of developers and the quality of the produced artefacts. While MDE is conceptually not restricted to a particular type of models, over the last few years approaches to MDE have converged on the automated management of models adhering to 3-layer metamodeling architectures, and most notably on models captured using the facilities provided by the the Eclipse Modelling Framework [2].

The majority of contemporary model management languages - including languages such as Acceleo, ATL, Kermeta, QVT and OCL - provide built-in support for EMF-based models. Using EMF as a de facto modelling framework has reduced unhelpful diversity and enhanced interoperability between MDE tools. However, in our view, for MDE tools to appeal to a wider audience of developers, they need to provide first-class support for other types of structured artefacts that developers commonly use to store meta-information. We first made this argument in [3], where we argued that providing support for schema-less XML documents would be beneficial for the wider adoption of MDE among software development practitioners. Our rationale is two-fold: first, meta-information description formats that do not require 3-layer architectures are generally simpler to learn and adopt, and can be used as a stepping stone for moving on to more powerful modelling architectures (e.g. EMF) once developers are convinced of the benefits of MDE. Second, there is already a significant amount of legacy meta-information captured in such formats for MDE researchers and tool providers to ignore.

Following this argument, in this work we have focused on providing support for integrating spreadsheets in MDE processes. Our intention is to enable engineers to use spreadsheets in the context of automated model management operations (such as model validation, transformation and code generation) in a *conceptually uniform manner to models*, i.e., that does not artificially impose transforming from/to an intermediate representation format (e.g. transforming behind the scenes spreadsheets to EMF models and vice versa).

The first challenge of providing support for managing spreadsheets with contemporary model management tools and languages lies in the different metaphors used by the two. Model management languages – influenced by 3-layer metamodelling architectures – typically provide an object-oriented syntax for manipulating information organised in terms of objects and relationships, whereas spreadsheets are structured in terms of worksheets, rows and columns. For spreadsheets to be manipulable by contemporary model management tools and languages, this gap needs to be bridged.

The other challenge is related to efficient querying of spreadsheets. Spreadsheet management tools typically provide highly efficient built-in query functionality (e.g. find all rows in worksheet X where the value of the second column is larger than Y) which need to be reused in model management languages. The alternative – encoding spreadsheet queries using naive iterations and comparisons in model management languages – is undesirable (in terms of reimplementation effort) and likely inefficient.

3 Querying and Modifying Spreadsheets using OCL-based Languages

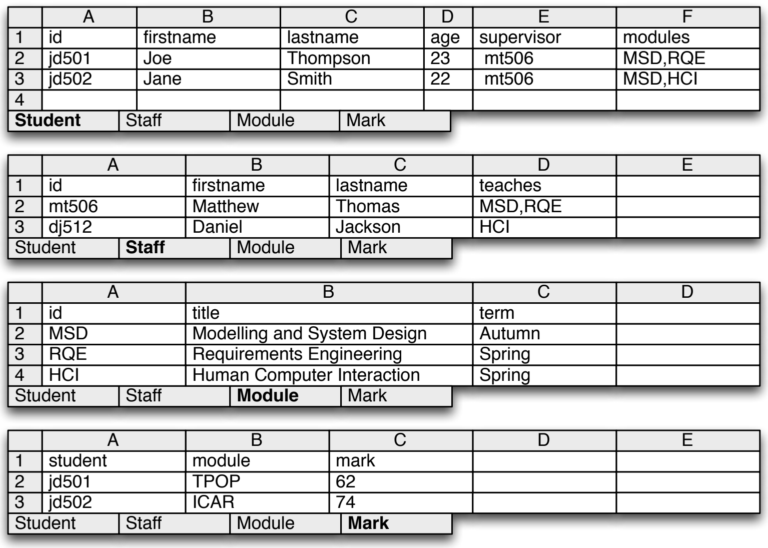
In this section we propose an approach for addressing these challenges at a conceptual level. Our approach comprises two separate aspects: querying and modifying spreadsheets. As such, it is in principle applicable both to side-effect free languages such as OCL and to transformation languages such as ATL [4], Kermeta [5], QVTo and EOL [6].

3.1 Querying Spreadsheets

Spreadsheets are organised into multiple tabular worksheets. Each worksheet typically has a name and a theoretically unbounded number of rows (numbered from 1 to ∞) and columns (titled incrementally using letters of the latin alphabet i.e. $A, B, \ldots, AA, AB \ldots, AAA, \ldots$). To bridge the gap between this data organisation paradigm and the object-oriented style of OCL-based model management languages we propose using worksheet names as (meta-)class names, and the values of the first cell of each column (e.g. A1, B1 etc.) as property names. Using only these assumptions, the spreadsheet illustrated in Figure 1, [1] can in principle be unambiguously queried using the following OCL expression to retrieve all students over the age of 25.

```
Student.allInstances->select(p:Student|p.age > 25)
```

[1] In this section, for simplicity, we use an artificial example to demonstrate the challenges and the proposed solutions for bridging the gap between spreadsheets and OCL-based languages. In the evaluation section, we demonstrate using the proposed approach with a requirements traceability spreadsheet.



	A	B	C	D	E	F
1	id	firstname	lastname	age	supervisor	modules
2	jd501	Joe	Thompson	23	mt506	MSD,RQE
3	jd502	Jane	Smith	22	mt506	MSD,HCI
4						

Student Staff Module Mark

	A	B	C	D	E
1	id	firstname	lastname	teaches	
2	mt506	Matthew	Thomas	MSD,RQE	
3	dj512	Daniel	Jackson	HCI	

Student **Staff** Module Mark

	A	B	C	D
1	id	title	term	
2	MSD	Modelling and System Design	Autumn	
3	RQE	Requirements Engineering	Spring	
4	HCI	Human Computer Interaction	Spring	

Student Staff **Module** Mark

	A	B	C	D	E
1	student	module	mark		
2	jd501	TPOP	62		
3	jd502	ICAR	74		

Student Staff Module **Mark**

Fig. 1. Spreadsheet comprising 4 worksheets: *Staff*, *Student*, *Module* and *Mark*

While using the first cell of each column as its name is straightforward if the engineer has control of the layout of the spreadsheet, it may not be an option for existing spreadsheets with a fixed layout. In terms of support for datatypes, most spreadsheet software support defining datatypes (*formats*) at the level of individual cells. When capturing data organised in a manner similar to that of Figure 1, it is useful to be able to specify the data types of entire columns (e.g. values in column D of the Student worksheet should be integers).

To enable engineers to decorate spreadsheets with such complementary information (column names, types etc.) that cannot be captured on the spreadsheet itself, we propose using an optional configuration model which conforms to the metamodel of Figure 2.

Spreadsheet configuration models can be constructed manually to complement/override the information (worksheet names, column names) already provided in spreadsheets of interest, or to generate automatically from object-oriented specifications (e.g. UML class diagrams or Ecore metamodels) via model-to-model transformation. The following paragraphs briefly discuss the roles, features and relationships of the concepts that comprise the spreadsheet configuration metamodel of Figure 2.

Worksheet Each worksheet can have an optional name (if a name is not provided, the name of the worksheet on the spreadsheet is used) and acts as a container for *Column* elements.

Column Each *Column* needs to specify its index in the context of the worksheet it belongs to, and optionally, a *name* (if a name is not provided, the one specified in its first cell is used as discussed above), an *alias*, a *datatype*, a *cardinality*, and

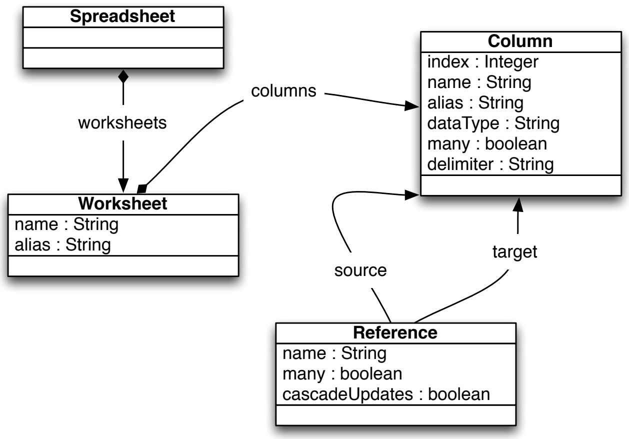


Fig. 2. Spreadsheet Configuration Metamodel

in case of columns with unbounded cardinality, the *delimiter* that should be used to separate the values stored in a single cell (comma is used as the default delimiter).

Reference In a configuration model engineers can specify ID-based references to capture relationships between columns belonging to potentially different worksheets. Each reference has a *source* and a *target* column, an optional *name* (if a name is not specified, the name of the source column is used to navigate the reference), a cardinality (*many* attribute), and specifies whether updates to cells of the target column should be propagated automatically (*cascadeUpdates* attribute) to the respective cells in the source column to preserve referential integrity.

For the spreadsheet illustrated in Figure 1, a single-valued reference can be defined between the contents of the *supervisor* column of worksheet *Student* and the *id* column of worksheet *Staff*, and a multi-valued reference can be defined between the *modules* column of the *Student* worksheet, and the *id* column of the module worksheet. Navigating a single-valued reference should return a row (or null), while navigating a multi-valued reference should return a non-unique ordered collection of rows to facilitate concise navigation over the spreadsheet data. Under these assumptions, to find all students whose supervisor name is *Thomas*, the following OCL query can be used.

```
Student.allInstances->select
  (s:Student|s.supervisor.firstname = "Thomas")
```

On a more complicated example, to find all modules taught by a member of staff called *Daniel*, the following query can be used.

```
Module.allInstances->select(m:Module|
  Staff.allInstances->exists(s:Staff|
    s.firstname="Daniel" and s.teaches->includes(m)))
```

3.2 Modifying Spreadsheets

While OCL itself is side-effect-free, many languages that build atop it – technically or conceptually – (such as QVTo, Kermeta, ATL and EOL) need to produce side-effects to support tasks such as model transformation and refactoring. In the context of supporting the requirements of such programs, three types of edit operations are required: creating and deleting rows, and modifying the values of individual cells. In this section we assume that languages capable of producing side-effects provide an additional assignment operator (:=), support for defining typed variables (e.g. through a *var* keyword), support for instantiating meta-types (e.g. through a *new* keyword) and deleting model elements (e.g. through a *delete* keyword), and built-in operations for modifying the contents of collections (e.g. add(), remove()).

Creating Rows As discussed above, in the proposed approach worksheets are treated as meta-classes and rows as their instances. As such, to create a new row in the Student worksheet, the meta-class instantiation capabilities of the action language can be used as follows. Creating a new row should not have any other side-effects on the spreadsheet.

```
var student : new Student;
```

Deleting Rows To delete a row from a worksheet, the respective syntax for deleting model elements in the action language can be used. When a row is deleted, all the rows that contain cells referring to it through cascade-update references also need to be recursively deleted.

```
var student = Student.allInstances->select(s:Student|s.id = "
    js502")->first();
delete student;
// deletes row 2 of the Student worksheet
// also deletes row 3 of the Mark worksheet
```

Modifying Cell Values If a cell is single-valued, a type-conforming assignment can be used to edit its value. For example, the following listing demonstrates modifying the age and the supervisor of a particular student.

```
var student : Student = ...;
var supervisor : Staff = ...;
student.age := 24;
student.supervisor := supervisor;
```

If on the other hand the cell is multi-valued, then its values should be handled as a collection. For example to move a module between two members of staff, the module row would need to be retrieved first, so that it can be removed/added from/to the *teaches* collections of the appropriate members of staff.

```
// Moves a module between two members of staff
var from : Staff := ...;
var to : Staff := ...;
var module : Module := ...;
from.teaches->remove(module);
to.teaches->add(module);
```

Updating the value of a cell can have side effects to other cells that are linked to it through cascade-update references to preserve referential integrity. For example, updating the value of cell A3 in the Module worksheet, should trigger appropriate updates in cells D2 and F2 of the Staff and Student worksheets respectively[2].

3.3 Efficient Querying

Using what has been discussed so far, to find all adult students in our spreadsheet, the following OCL query would need to be constructed and evaluated.

```
Student.allInstances->select(s:Student | s.age > 17);
```

Evaluating this query in a naive manner would involve iterating through all the rows of the Student worksheet, retrieving the value of the third cell of each row, casting it to an integer and comparing it against the predefined value. For large spreadsheets, this would be sub-optimal, particularly given that most spreadsheet management systems provide built-in search capabilities. For example, the OCL query above can be expressed in the Google Spreadsheet query language as follows:

```
https://spreadsheets.google.com/feeds/list/tb-
<student-worksheet-guid>/od6/private/full?sq=age>17
```

To support efficient execution of simple queries, we propose detecting optimisable patterns and rewriting them as native queries instead where possible (one such pattern is displayed below). The potential for optimisation through rewriting and the details of the rewriting process predominately depend on the expressiveness and the operators supported by the native query language.

```
X.allInstances->select(x:X | x.p1 = y)
```

4 Prototype

To evaluate the feasibility and practicality of the proposed approach, we have implemented a prototype that adds support for managing Google Spreadsheets to the Epsilon platform of model management languages [3]. Below, we briefly discuss the relevant Epsilon infrastructure and then illustrate the Google Spreadsheets extension.

[2] We intentionally refrain from any further discussion on cascade-update algorithms as this is a trivial and well-understood topic.

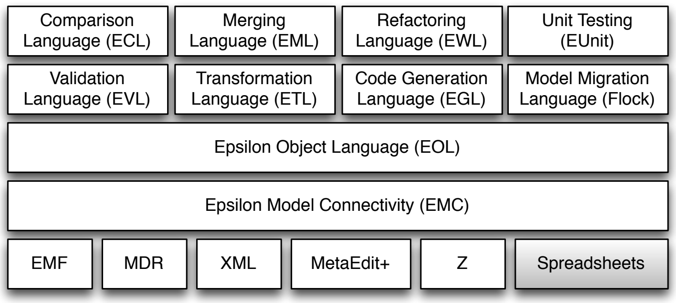
[3] The prototype is available under http://epsilon-emc-google-spreadsheet-driver.googlecode.com/

4.1 Epsilon

Epsilon [7] is a mature family of interoperable languages for model management. Languages in Epsilon can be used to manage models of diverse metamodels and technologies (detailed below). The core of Epsilon is the Epsilon Object Language (EOL) [6], an OCL-based imperative language that provides additional features including model modification, multiple model access, conventional programming constructs (variables, loops, branches etc.), user interaction, profiling, and support for transactions. EOL can and has been used as a general-purpose model management language (e.g. for operational model transformation). It is primarily intended to be reused in task-specific model management languages. A number of task-specific languages have been implemented atop EOL, including: model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL) and model-to-text transformation (EGL). These languages reuse EOL in different ways, e.g. by acting as a preprocessor, or by using EOL to define behaviour of rules.

Epsilon is designed to be technology agnostic - that is, the same Epsilon program can be used to manage models from different technologies: the concepts and tasks of model management are independent of how models are represented and stored. To support this, Epsilon provides the Epsilon Model Connectivity (EMC) layer[4], which offers a uniform interface for interacting with models of different modelling technologies. New technologies are supported by adding a *driver* to EMC. Currently, EMC drivers have been implemented to support EMF [2] (XMI 2.x), MDR [8] (XMI 1.x), pure XML, and Z [9] specifications in LaTeX using CZT [10] Also, to enable users to compose complex workflows that involve a number of individual model management tasks, Epsilon provides ANT [11] tasks and an inter-task communication framework discussed in detail in [12].

The technical architecture of Epsilon is illustrated in Figure 3.



Comparison Language (ECL)
Merging Language (EML)
Refactoring Language (EWL)
Unit Testing (EUnit)
Validation Language (EVL)
Transformation Language (ETL)
Code Generation Language (EGL)
Model Migration Language (Flock)
Epsilon Object Language (EOL)
Epsilon Model Connectivity (EMC)
EMF
MDR
XML
MetaEdit+
Z
Spreadsheets

Fig. 3. Overview of the architecture of Epsilon

As mentioned earlier, EMC enables developers to implement *drivers* – essentially classes that implement the `IModel` interface of Figure 4 – to support diverse modelling technologies. The work in this section illustrates the design and

[4] `http://www.eclipse.org/epsilon/doc/emc`

implementation of a new driver (in addition to the existing drivers for managing EMF, MDR and Z, XML models) for interacting with Google Spreadsheets.

In addition to abstracting over the technical details of specific modelling technologies, EMC facilitates the concurrent management of models expressed with different technologies. For instance, Epsilon can be used to transform an EMF-based model into an MDR-based model, to perform inter-model validation between a Z model and an EMF model, or to develop a code generator that consumes information from an EMF-based and a Google Spreadsheet model at the same time.

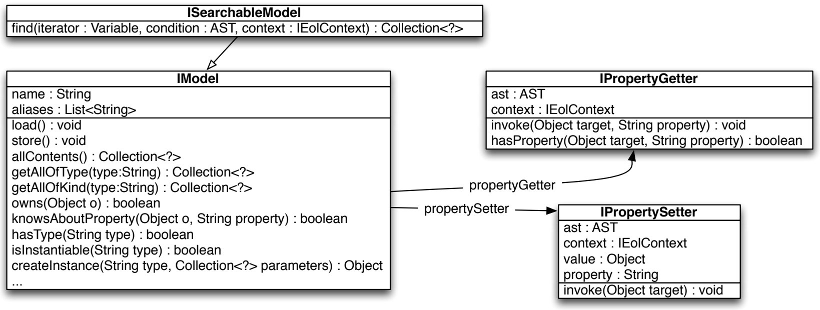

ISearchableModel
find(iterator : Variable, condition : AST, context : IEolContext) : Collection<?>
IModel
name : String
aliases : List<String>
load() : void
store() : void
allContents() : Collection<?>
getAllOfType(type:String) : Collection<?>
getAllOfKind(type:String) : Collection<?>
owns(Object o) : boolean
knowsAboutProperty(Object o, String property) : boolean
hasType(String type) : boolean
isInstantiable(String type) : boolean
createInstance(String type, Collection<?> parameters) : Object
...
IPropertyGetter
ast : AST
context : IEolContext
invoke(Object target, String property) : void
hasProperty(Object target, String property) : boolean
propertyGetter
propertySetter
IPropertySetter
ast : AST
context : IEolContext
value : Object
property : String
invoke(Object target) : void

Fig. 4. Epsilon Model Connectivity (EMC) Layer Interfaces

4.2 The EMC Google Spreadsheet Driver

As illustrated in Figure 5, support for Google Spreadsheets has been implemented as a driver conforming to the EMC interfaces. More specifically, the *GSModel* class acts as a wrapper for Google Spreadsheets and implements methods such as *getAllOfType(String type)* which returns all the rows of a particular worksheet (type), and *createInstance(String type)/deleteElement(Object element)* which can create and delete rows respectively. Classes *GSPropertyGetter* and *GSPropertySetter* on the other hand are responsible for retrieving and setting property values of rows.

As displayed in Figure 5, we have used an additional level of abstraction between the EMC interfaces and their Google Spreadsheet implementations which capture spreadsheet-specific but Google Spreadsheet-independent logic and which can be reused to implement support for additional types of spreadsheets (e.g. Microsoft Excel or Open Office spreadsheets).

The Google Spreadsheet driver adopts a lazy approach to retrieving data from remote spreadsheets instead of attempting to construct an in-memory copy of the entire contents of the spreadsheet upon initialisation. Also, all side-effects produced on spreadsheets are propagated directly to the remote spreadsheet to avoid the problem of working with stale data.

With regard to spreadsheet configuration models, we have opted for an XML-based concrete syntax in an effort to enable engineers to use this driver with minimal effort (using EMF-based configuration models would be a technically

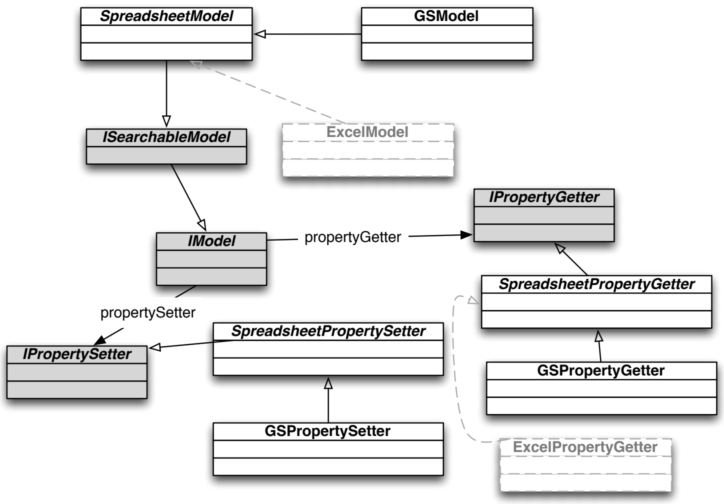


Fig. 5. Google Spreadsheet EMC Driver Design

more sound approach but would require engineers to become familiar with EMF). The supported datatypes in the spreadsheet configuration models are: *integer*, *boolean*, *String*(default), *real* and *double*. Using this XML-based format, the configuration model for the spreadsheet of Figure 1 is illustrated below.

```
<spreadsheet>
  <worksheet name="Student">
    <column name="age" datatype="integer"/>
  </worksheet>
  <worksheet name="Mark">
    <column name="mark" datatype="integer"/>
  </worksheet>
  <worksheet name="Staff">
    <column name="teaches" many="true" delimiter=","/>
  </worksheet>
  <reference source="Student->supervisor"
          target="Staff->id"/>
  <reference source="Staff->teaches"
          target="Module->id" many="true"/>
  ...
</spreadsheet>
```

With regard to performing efficient queries on spreadsheets, *GSModel* implements the *ISearchableModel* interface provided by EMC, and implements its *find(Variable iterator, AST condition)* method in which it rewrites optimisable conditions as native Google Spreadsheet queries at runtime as discussed in Section 3. The rewriter employs a recursive descent algorithm, and fully exploits the capabilities of the Google Spreadsheet query language (numeric and string comparisons as well as composite queries consisting of more than one and/or clauses). Non-optimisable conditions in *find* cause the driver to fail gracefully.

The following listing demonstrates using the *find* operation to perform queries on the spreadsheet of Figure 1.

```
// Collects all the first names
// of students with marks < 50
M->find(m:Mark | m.mark < 50)->
  collect(m:Mark | m.student.firstname);

// Fails gracefully as the condition involves
// a two-level property navigation and cannot be
// rewritten as a native query
M->find(s:Student |
  s.supervisor.firstname = "Daniel");
```

By providing a Google Spreadsheet driver for EMC, all languages that build atop EOL can now interact with such spreadsheets. For example, the EVL constraint below checks that no member of staff teaches more than 4 modules. This is illustrated further in the case study that follows.

```
context Staff {
  constraint NotOverloaded {
    check: self.teaches->size() <= 4
    message: "Member of staff" + self.firstname +
      " " + self.lastname + " is overloaded"
  }
}
```

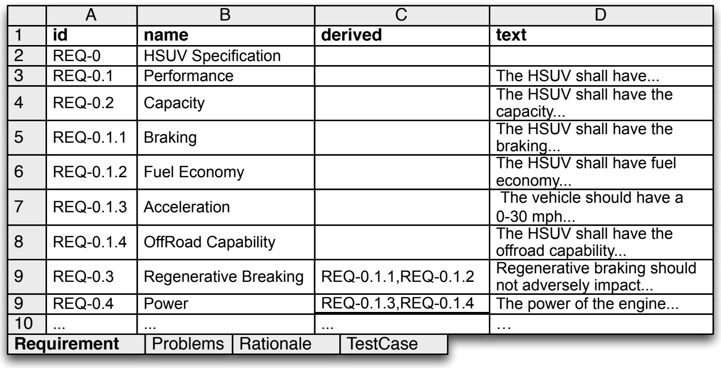
5 Case Study

In this section we demonstrate the proposed approach by applying it to representative case study. This case study is provided in the official SysML documentation [13], and is based on a specification published by the National Highway Traffic Safety Administration (NHTSA).

5.1 Hybrid SUV Example

The case study illustrates the application of the proposed approach for the development of a Hybrid gas/electric powered Sport Utility Vehicle (SUV). It is interesting in that it consists of a complex requirements hierarchy which can be captured efficiently using spreadsheets. The example focuses mainly on the requirements engineering phase of the development process and how this can benefit from treating spreadsheets as models. A significant benefit comes from the seamless integration of spreadsheets with downstream MDE tasks.

In this case study, the various system requirements are captured in a spreadsheet comprising four worksheets. These requirements concern the operation and performance of the vehicle. Figure 6 illustrates an excerpt of the requirements spreadsheet of the Hybrid SUV. In the first worksheet the system requirements are captured. The first column captures the unique requirement identifiers. These identifiers have a fixed format and they conform to the dot notation. The subsequent columns capture the name of a requirement, whether a requirement is derived from another requirement and finally the requirement's text.



	A	B	C	D
1	**id**	**name**	**derived**	**text**
2	REQ-0	HSUV Specification		
3	REQ-0.1	Performance		The HSUV shall have...
4	REQ-0.2	Capacity		The HSUV shall have the capacity...
5	REQ-0.1.1	Braking		The HSUV shall have the braking...
6	REQ-0.1.2	Fuel Economy		The HSUV shall have fuel economy...
7	REQ-0.1.3	Acceleration		The vehicle should have a 0-30 mph...
8	REQ-0.1.4	OffRoad Capability		The HSUV shall have the offroad capability...
9	REQ-0.3	Regenerative Breaking	REQ-0.1.1,REQ-0.1.2	Regenerative braking should not adversely impact...
9	REQ-0.4	Power	REQ-0.1.3,REQ-0.1.4	The power of the engine...
10	...	...	...	...

Requirement | Problems | Rationale | TestCase

Fig. 6. Hybrid SUV requirements spreadsheet.

5.2 Managing Requirements Spreadsheets with Epsilon

In this section, we illustrate how the proposed approach can be used in the context of the Hybrid SUV case study. More specifically, we show three scenarios: how we can query the spreadsheet, how we can validate the correctness of the information captured in the spreadsheet and, finally, how we can generate textual artefacts from it. The spreadsheet configuration model for this spreadsheet follows.

```
<spreadsheet>
  <reference source="Requirement->derived"
    target="Requirement->id"/>
  <worksheet title="Requirement">
    <column title="derived" many="true" delimiter=","/>
  </worksheet>
</spreadsheet>
```

Querying Requirements Spreadsheets In the first usage scenario, the engineer wishes to retrieve all the children in the requirement hierarchy for a given requirement. By doing this, the engineer wishes to understand the rationale behind a given composite requirement and how this composite requirement is decomposed. In the proposed approach, to do this the engineer has to define an EOL operation, which returns a collection with all the children elements of the hierarchy for a given requirement. This operation is illustrated in Listing 1.1.

```
operation Requirement getChildren() : Sequence {
  var children : new Sequence;
  //iterates all requirements, and checks if their id
  //starts with the id of the parent
  for(r in Requirement.all) {
    if((r.id+'.').startsWith(self.id+'.')){
      children.add(r);
    }
  }
  //returns a set with all the requirements
  //in the hierarchy
  return children;
}
```

Listing 1.1. getChildren() EOL operation

To identify the child requirements, the operation relies on the requirement ids and the convention that all the requirement identifiers have to conform to the dot notation. Therefore, this operation assumes that all the requirements in the spreadsheet have a valid id. In the following we will demonstrate how an engineer could check if these assumptions hold.

Validating Requirements Spreadsheets To validate the contents of the requirements spreadsheet the engineer can specify a set of EVL constraints. These constraints are illustrated in Listing 1.2; if the constraints hold, all the requirements have an id and they all have a valid format. To simplify the expression of these constraints, we also write an EOL operation (Listing 1.3), which encapsulates the functionality that checks the correctness (using a regular expression) of an id's format.

```
// For all requirements
context Requirement {
  //Checks whether the requirement has an id
  constraint HasId {
    check: self.id.isDefined()
    message: 'Requirement ' + self.name + 'does not have an id.'
  }
  //Checks whether the requirement has a a valid id
  constraint HasValidId {
    check: self.hasValidId()
    message: 'Requirement ' + self.name + 'does not have a valid
        id.'
  }
}
```

Listing 1.2. EVL constraints for requirements spreadsheets

```
// This operation uses a regular expression to
// test whether the requirement id has a valid format
operation Requirement hasValidId() : Boolean {
  return self.id.matches("REQ-[0-9]+(\\.([0-9]+))*"));
}
```

Listing 1.3. Requirement ID validation operation

Generating Textual Artefacts Once the requirements are well understood and well-formatted they can be used as first-class citizens in the MDE development process. For instance, a requirements derivation graph such as the one displayed in Figure 7 can be generated automatically using GraphViz [14] and the EGL model-to-text transformation below.

```
digraph G {
  [%for (r in Requirement.all) { %]
  [%for (derived in r.derived) { %]
  "[%=r.id%]" -> "[%=derived.id%]";
  [%}%]
  [%}%]
}
```

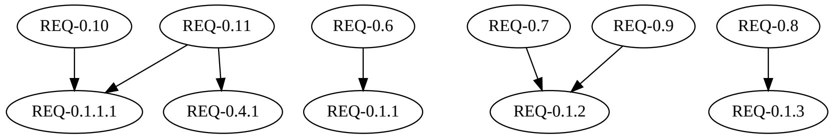


Fig. 7. Hybrid SUV requirements derivation graph.

Additional MDE tasks and operations can now easily be carried out, e.g. model-to-model transformations to more structured formats (such as EMF models), or update-in-place transformations to correct any 'bad smells' [15] that can be identified.

6 Related Work

Although spreadsheets are widely used in software engineering for supporting numerous development activities (such as collecting system requirements, monitoring project process and capturing traceability information) there is little research on using them formally in a software engineering lifecycle. The majority of the literature focuses mainly on how to use software engineering principles to support the development of high quality spreadsheets. There are two main approaches to spreadsheet engineering - constructive and analytical [17]. The purpose of the former is to ensure that spreadsheets are correct by construction, while the latter aims to detect errors after a spreadsheet has been created.

Two representative examples of an analytical approach are [18] and [19]. The former uses data flow adequacy criteria and coverage monitoring to test spreadsheets, while the latter is an extension which employs fault localisation techniques to isolate spreadsheet errors. In [20] user-provided assertions about cell ranges are used to identify errors in formulas. The FFR (formulae, formats, relations) model [21] abstracts the structure and fundamental features of spreadsheets without paying attention to the detailed semantics of operations and functions. Anomalies in the structure of the model (for example breaking areas) highlight possible errors. In [15], analogies to code 'smells' are specified and analysed against spreadsheets and worksheets to identify flaws and design errors; their experiments show that numerous errors can be automatically identified.

ClassSheet [17] is the most popular constructive approach. This approach introduces a formal higher-level object-oriented model which can be used to automatically generate spreadsheets. Given the formal nature of this model a number of typical errors associated with spreadsheets can be prevented, as the spreadsheets will be correct by construction. A more recent constructive approach to spreadsheet engineering is *MDSheet* [22]. This is a model-driven approach and it is based on *ClassSheet*. *MDSheet* relies on a bi-directional transformation framework [23] in order to maintain spreadsheet models (i.e. *ClassSheet* models) and their instances synchronized.

The focus of the aforementioned research work is quite different from the focus of the proposed approach. As mentioned previously, past research focuses on how to ensure the correctness of spreadsheets. On the other hand, this paper proposes an approach whose focus is on the seamless integration of spreadsheets in the MDE development process, as well as their elevation to first-class models. In our approach spreadsheets are considered just another type of model in an MDE process. As such, spreadsheets can be queried, validated or even transformed to other artefacts using MDE techniques and concepts.

7 Conclusions and Further Work

In this paper, we have argued for the importance of adding support for spreadsheets to the MDE toolkit and in particular to enable their support in OCL-based model management languages. We have presented an approach that bridges the conceptual gap between the tabular nature of spreadsheets and the object-oriented nature of contemporary modelling technologies and model management languages, which also addresses the problem of efficiently querying spreadsheets from within such languages. We have evaluated this approach by constructing a prototype Google Spreadsheet driver on top of the model connectivity framework of Epsilon. We have also presented a case study that demonstrates the practicality and usefulness of managing spreadsheets with model management languages.

In future iterations of this work, we plan to extend the spreadsheet driver to target additional types of spreadsheets, and construct transformations that can generate spreadsheet configuration models from object-oriented specifications such as UML class models and Ecore metamodels. We also intend to use model refactoring languages and tools (particularly EWL) to support spreadsheet and worksheet refactoring to automatically eliminate so-called ‘bad smells’ [15, 24].

References

1. Donald Firesmith. Common requirements problems, their negative consequences, and industry best practices to help solve them. *in Journal of Object Technology*, 6:17–33, 2007.
2. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modelling Framework*. Eclipse Series. Addison-Wesley Professional, 2 edition, December 2008.
3. Dimitrios S. Kolovos, Louis M. Rose, James Williams, Nicholas Matragkas, and Richard F. Paige. A lightweight approach for managing xml documents with mde languages. In *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 2012.
4. Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.
5. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278, Berlin, Heidelberg, 2005. Springer-Verlag.
6. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
7. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
8. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.
9. Jim Woodcock and Jim Davies. *Using Z : Specification, Refinement, and Proof.* Prentice Hall, March 1996.
10. Community Z Tools. http://czt.sourceforge.net.
11. The Apache Ant Project. http://ant.apache.org.
12. Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *Proc. Workshop on Model Driven Tool and Process Integration (MDTPI), ECMDA*, Berlin, Germany, June 2008.
13. OMG, Systems Modeling Language (SysML). http://www.omg.org/spec/SysML/1.3/PDF/, 2012.
14. Graphviz - Graph Visualization Software, Official Web-Site. http://www.graphviz.org.
15. Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *ICSE*, pages 441–451. IEEE, 2012.
16. Raymond R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. *Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, July 2000.
17. Gregor Engels and Martin Erwig. Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 124–133, New York, NY, USA, 2005. ACM.
18. Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol.*, 10(1):110–147, January 2001.

19. S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and behaviors of end-user programmers with interactive fault localization. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, HCC '03, pages 15–22, Washington, DC, USA, 2003. IEEE Computer Society.
20. Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 93–103, Washington, DC, USA, 2003. IEEE Computer Society.
21. Jorma Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages & Computing*, 11(1):49 – 82, 2000.
22. Jácome Cunha, João P. Fernandes, Hugo Ribeiro, and João Saraiva. Mdsheet: A framework for model-driven spreadsheet engineering. In *Software Engineering (ICSE), 34th International Conference on*, pages 1395–1398, June 2012.
23. Jácome Cunha, João P. Fernandes, Hugo Ribeiro, and João Saraiva. A bidirectional model-driven spreadsheet environment. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1443–1444, June.
24. Jácome Cunha, João P. Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part IV*, ICCSA'12, pages 202–216, Berlin, Heidelberg, 2012. Springer-Verlag.